

FAST HIERARCHICAL SOLVERS FOR SPARSE MATRICES USING LOW-RANK APPROXIMATION*

HADI POURANSARI[†], PIETER COULIER^{†‡}, AND ERIC DARVE^{†§}

Abstract. Inversion of sparse matrices with standard direct solve schemes is robust, but computationally expensive. Iterative solvers, on the other hand, demonstrate better scalability; but, need to be used with an appropriate preconditioner (e.g., ILU, AMG, Gauss-Seidel, etc.) for proper convergence. The choice of an effective preconditioner is highly problem dependent. We propose a novel fully algebraic sparse matrix solve algorithm, which has linear complexity with the problem size. Our scheme is based on the Gauss elimination. For a given matrix, we approximate the LU factorization with a tunable accuracy determined a priori. This method can be used as a stand-alone direct solver with linear complexity and tunable accuracy, or it can be used as a black-box preconditioner in conjunction with iterative methods such as GMRES. The proposed solver is based on the low-rank approximation of fill-ins generated during the elimination. Similar to \mathcal{H} -matrices, fill-ins corresponding to blocks that are well-separated in the adjacency graph are represented via a hierarchical structure. The linear complexity of the algorithm is guaranteed if the blocks corresponding to well-separated clusters of variables are numerically low-rank.

Key words. sparse, hierarchical, low-rank, elimination, compression

AMS subject classifications. 65F05, 65F08, 65F50, 65N55, 68Q25

1. Introduction. In the realm of scientific computing, solving a sparse linear system,

$$(1.1) \quad A\mathbf{x} = \mathbf{b},$$

is known to be one of the challenging parts of many calculations, and is often the main bottleneck. Such a system of equations may be the result of the discretization of some partial differential equation (PDE), or more generally, can represent the local interactions of units in a network.

Solving a system of equations of size n using a naive implementation of Gauss elimination has $\mathcal{O}(n^3)$ time complexity. The best proved time complexity to solve a general linear system is $\mathcal{O}(n^\omega)$, where $\omega < 2.376$ [12, 17, 40]. In the case of sparse matrices, the time and memory complexity can be reduced when a proper elimination order is employed. Finding the optimal ordering (that results in the minimum number of new non-zeros in the LU factorization) is known to be an NP-complete problem [57]. For matrices resulting from the discretization of some PDE in physical space, nested dissection [23, 43] is known as an efficient elimination strategy. [1] discusses the complexity of nested dissection based on the sparsity pattern of the matrix. For a three-dimensional problem, the time and memory complexities are expected to be $\mathcal{O}(n^2)$ and $\mathcal{O}(n^{4/3})$, respectively, when nested dissection is employed. As the size of the problem grows, such complexities make direct solvers prohibitive.

*Funding from the “Army High Performance Computing Research Center” (AHPCRC), sponsored by the U.S. Army Research Laboratory under contract No. W911NF-07-2-0027, at Stanford, supported in part this research. The second author is a post-doctoral fellow of the Research Foundation Flanders (FWO) and a Francqui Foundation fellow of the Belgian American Educational Foundation (BAEF). The financial support is gratefully acknowledged.

[†]Stanford University, Department of Mechanical Engineering, Stanford, CA 94305, USA ([hadip,pcoulier,darve]@stanford.edu).

[‡]KU Leuven, Department of Civil Engineering, Kasteelpark Arenberg 40, 3001 Leuven, Belgium.

[§]Stanford University, Institute for Computational and Mathematical Engineering, Stanford, CA 94305, USA.

Iterative methods, such as conjugate gradient [36], minimum residual [47], and general minimum residual [53], are generally more time and memory efficient. In addition, iterative solvers such as those based on Krylov subspace can be accelerated using fast linear algebra techniques. The fast multipole method (FMM) [20, 22, 28, 45, 49, 58], for example, can accelerate matrix-vector multiplication—from quadratic complexity to linear—which is the bulk calculation in iterative solvers based on Krylov subspace. However, in practice, iterative methods need to be used in conjunction with preconditioners to limit the number of iterations. The choice of an efficient preconditioner is highly problem dependent. There are many ongoing efforts to develop preconditioners that are optimized for particular applications. Hence, there is a need for general purpose preconditioners. Hierarchical matrices enable us to develop such preconditioners.

FMM matrices are a subclass of a larger category of matrices called hierarchical matrices (\mathcal{H} -matrices) [7, 9, 32]. \mathcal{H} -matrices have a hierarchical low-rank structure. For instance, in a hierarchically off-diagonal low-rank (HODLR) matrix [5], off-diagonal blocks can be represented through a hierarchy of low-rank interactions. If the bases used in the hierarchy are nested (i.e., the low-rank basis at each level is constructed using the low-rank basis of the child level) the method is called hierarchically semi-separable (HSS) [3, 15, 56]. In a more general case of hierarchical matrices, more complex low-rank structures can be considered. A full dense matrix with many low-rank structures is in fact data-sparse [31, 33]. A data-sparse matrix can be represented via an extended sparse matrix, which is larger in size—but of the same order as the original matrix—and has few non-zero entries [2]. The hierarchical structure of such matrices can be used for efficient calculation and storage.

Recently, hierarchical interpolative factorization [38, 39] was proposed, which can be used to directly solve systems obtained from differential and integral equations based on elliptic operators. The fast factorization is obtained by skeletonization of fronts in the multifrontal scheme. Using low-rank structure of the off-diagonal blocks to develop fast direct solvers for linear systems arising from integral equations has been widely studied [18, 26, 27, 41]. [25] proposed a direct solver for elliptical PDEs with variable coefficients on two-dimensional domains by exploiting internal low-rank structures in the matrices. [44] used hierarchical low-rank structures of the off-diagonal blocks to introduce a preconditioner for sparse matrices based on a multifrontal variant of sparse LU factorization. [46] introduced a black-box linear solver using tensor-train format. [42] used a recursive low-rank approximation algorithm based on Sherman-Morrison formula to obtain a preconditioner for symmetric sparse matrices.

Sparse matrices can be considered as a very special case of hierarchical matrices, where instead of low-rank blocks they initially have zero blocks. However, during the elimination process in a direct solve scheme, many of the zero blocks get filled. For a large category of matrices, including those obtained from the discretization of PDEs, most of the new fill-ins are numerically low-rank. This is justified when the Green's function associated to the PDE is smooth. In this paper, we will use the \mathcal{H} -matrix structure to compress the fill-ins. A similar process can be applied in the elimination of an extended sparse matrix resulting from an originally dense matrix [4, 19]. This reduces the complexity of the direct solver to linear. The linear complexity of the method is guaranteed if the blocks corresponding to the interaction of well-separated nodes are numerically low-rank. We define the well-separated condition in section 3.1.

The proposed algorithm can be considered as an extension to the block incom-

plete LU (ILU) [52] preconditioners. In a block ILU factorization, most of the new fill-ins (i.e., blocks that are created during the elimination process which are originally zero) are ignored, and therefore, the block sparsity pattern of the matrix is preserved, while the accuracy is not. In the proposed algorithm, instead, we use low-rank approximations to compress such new fill-ins. Using a tree structure, new fill-ins at the fine level are compressed and pushed to the parent (coarse) level. The elimination and compression processes are done in a bottom-to-top traversal.

In addition, the proposed algorithm has formal similarities with algebraic multi-grid (AMG) methods [10, 11, 51, 54]. However, the two methods differ in the way they build the coarse system, and use restriction and prolongation operators. In AMG, the original system is solved at different levels (from fine to coarse). Here, the compressed fill-ins—corresponding to the Schur complements—of each level are solved at the coarser level above. Note that the proposed algorithm is purely algebraic, similar to AMG. If the matrix comes from discretization of a PDE on a physical grid, the grid information can be exploited to improve the performance of the solver, similar to geometric multi-grid.

The algorithm presented in this paper computes an approximate inverse of a sparse matrix using Gauss elimination. We introduce intermediate operations to compress new fill-ins. The compressed fill-ins are represented using a set of extra variables. This technique is known as extended sparsification [14]. The algorithm has linear complexity with the size of the problem (when well-separated interactions are numerically low-rank) using hierarchical structures. The accuracy of the factorization phase (i.e., Gauss elimination and compression), ϵ , can be determined a priori. The time and memory complexity of the factorization are $\mathcal{O}(n \log^2 1/\epsilon)$ and $\mathcal{O}(n \log 1/\epsilon)$, respectively, as will be clarified in section 6.1.

The method presented in this paper is similar to the fast hierarchical methods developed by Hackbusch et al. [9, 31, 32, 33] in a sense that both methods use a tree decomposition to identify and represent low-rank blocks. The key difference, however, is that in the Hackbusch’s algorithm the LU factorization is computed using a depth-first tree traversal order, whereas here we use a breadth-first (level by level from leaf to root) traversal. The connection and differences of the proposed algorithm and Hackbusch’s fast \mathcal{H} -algebra is discussed in our companion paper [19], in which a similar method is used for dense matrix factorization.

Our solver can be used as a stand-alone direct solver with tunable accuracy. The factorization part is completely separate from the solve part and is generally more expensive. This makes the algorithm appealing when multiple right hand sides are available (e.g., using the proposed solver as a black-box preconditioner in an iterative method). We have implemented the algorithm in C++ (the code can be downloaded from bitbucket.org/hadip/lorasp), and benchmarked it as both a stand-alone solver (see section 6.1), and a preconditioner in conjunction with the generalized minimum residual (GMRES) iterative solver [53] (see section 6.2).

Furthermore, the proposed algorithm has interesting parallelization properties. On one hand, all calculations are block matrix computations which can be highly accelerated using BLAS3 operations [6]. On the other hand, since the sparsity pattern at every level is preserved, the data dependency is very local, which is an interesting property to reduce the amount of communications. In addition, the amount of calculation scales with the third power of the size of blocks, while the communications scales with the second power of block sizes. This helps with the concurrency of the parallel implementation. Moreover, the order of elimination does not change the

complexity of the presented algorithm. This is in particular an appealing property for parallel implementation. The parallel implementation of the proposed method is not further discussed in this paper.

The remainder of this paper is organized as follows. In section 2 we briefly introduce a graph representation of sparse matrices, and an interpretation of the Gauss elimination using the adjacency graph. In section 3 some concepts related to the hierarchical representation of matrices are defined. The algorithm is explained in section 4 in detail, and the linear complexity analysis is provided in section 5. We present numerical results obtained from various benchmarks in section 6. The proposed algorithm is a general framework that provides a sparse matrix direct factorization. The factorization has linear time and memory complexities if well-separated blocks of the matrix have a low-rank interaction. There are many avenues for optimization and extension of the algorithm. We discuss some of these opportunities in section 7.

2. Sparse linear systems. In this section we briefly introduce the graphical framework that is required in the rest of the paper. We assume a sparse linear system $A\mathbf{x} = \mathbf{b}$ of size n is given.

2.1. Adjacency graph. In many algorithms, including the method proposed in this paper, it is necessary (or more efficient) to operate on sub-blocks of the matrix rather than single elements. The blocks of the matrix can be identified using a partitioning as defined below.

DEFINITION 2.1. (*partitioning*) A partitioning \mathcal{P} is defined as a map $\{1, \dots, n\} \rightarrow \{1, \dots, n_b\}$. Essentially, \mathcal{P} maps each row/column index i of the matrix to a block index $\mathcal{P}(i)$, where there are n_b blocks.

For $1 \leq \mathbf{i} \leq n_b$, and $1 \leq \mathbf{j} \leq n_b$, we use $A_{\mathbf{i},\mathbf{j}}$ to represent all entries $A_{k,t}$ such that $\mathcal{P}(k) = \mathbf{i}$ and $\mathcal{P}(t) = \mathbf{j}$.

It is often fruitful to represent sparse linear systems using graphs. An adjacency graph, as defined below, represents a sparse matrix with block-partitioning.

DEFINITION 2.2. (*adjacency graph*) A sparse matrix A with a partitioning \mathcal{P} to n_b blocks can be represented by a directed graph $G(V, E)$, where $V = \{v_1, \dots, v_{n_b}\}$. Each $v_{\mathbf{i}} \in V$ for $1 \leq \mathbf{i} \leq n_b$ represents a set of columns (variables) and a set of rows (equations) corresponding to the diagonal block $A_{\mathbf{i},\mathbf{i}}$. A vertex $v_{\mathbf{i}}$ is connected to a vertex $v_{\mathbf{j}}$ by an edge $e_{\mathbf{i} \rightarrow \mathbf{j}} \in E$ if and only if the block $A_{\mathbf{j},\mathbf{i}}$ in A is non-zero. G is the adjacency graph of the matrix A with partitioning \mathcal{P} .¹

In the rest of the paper we use vertex and node interchangeably for the adjacency graph. In fig. 1, an example of the adjacency graph is illustrated. For a node p in the adjacency graph, we denote its vector of variables by $\mathbf{var}(p)$. Also, $\mathbf{rhs}(p)$ denotes the vector of right hand sides corresponding to the equations associated to the node p . For the example shown in fig. 1, the set of equations corresponding to the node p can be written as follows.

$$A_{p,p}\mathbf{var}(p) + A_{p,c}\mathbf{var}(c) + A_{p,o}\mathbf{var}(o) + A_{p,g}\mathbf{var}(g) = \mathbf{rhs}(p)$$

¹The adjacency graph of a matrix A with a partitioning \mathcal{P} is essentially the *quotient graph* of the adjacency graph of the matrix A with identity partitioning, where the equivalence relation is induced by partitioning \mathcal{P} .

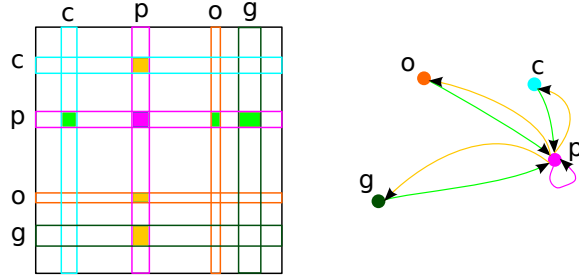


FIG. 1. Example of the adjacency graph (right) of a sparse matrix (left). Vertices' colors (c , p , o , and g) are the same as their corresponding rows/columns in the matrix. Edges' colors are also in correspondence with the sub-blocks in the matrix.

2.2. Elimination. The standard Gauss elimination process, or the LU factorization, can also be explained using the graph representation of the matrix. At each step of the elimination process, a set of unknowns is eliminated from the system of equations. This corresponds to eliminating a vertex from the adjacency graph G . Assume that we are about to eliminate a vertex v_i from the graph. The self-edge from v_i to itself corresponds to the pivot diagonal sub-block in the matrix. After eliminating v_i , for every pair of outgoing edge $e_{i \rightarrow j}$ to a vertex v_j and incoming edge $e_{k \rightarrow i}$ from a vertex v_k , a new edge from v_k to v_j is created, corresponding to the Schur complement of the eliminated edges, that is $-A_{j,i}A_{i,i}^{-1}A_{i,k}$. Note that if the edge between v_k and v_j exists before elimination, the Schur complement adds to the existing sub-block.

The process described above reveals the fact that during the elimination process many new edges are introduced in the graph. This corresponds to generating new non-zero blocks in the matrix during the LU factorization. The generation of many dense blocks is what makes the direct factorization of sparse matrices a prohibitive process. Essentially, a matrix A can be sparse, while L and U in the LU factorization of A are dense. In the next section, we explain how we can preserve the sparsity pattern of the matrix during the elimination process using the compression of well-separated interactions.

2.3. Key idea. An important observation in the elimination process is the fact that fill-ins (i.e., new edges created during the elimination process) that correspond to well-separated vertices are often numerically low-rank. For a linear system obtained from a discretized PDE, well-separated refers to nodes that are physically far enough from each other. For a general sparse matrix two vertices are well-separated if their distance in the adjacency graph is large enough. It is formally defined in definition 3.6. We replace such fill-ins with a sequence of low-rank operations.

For example, consider the following symmetric linear system that is partitioned into 3 blocks

$$(2.1) \quad \begin{pmatrix} S & B & C \\ B^\top & P & \\ C^\top & & Q \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}.$$

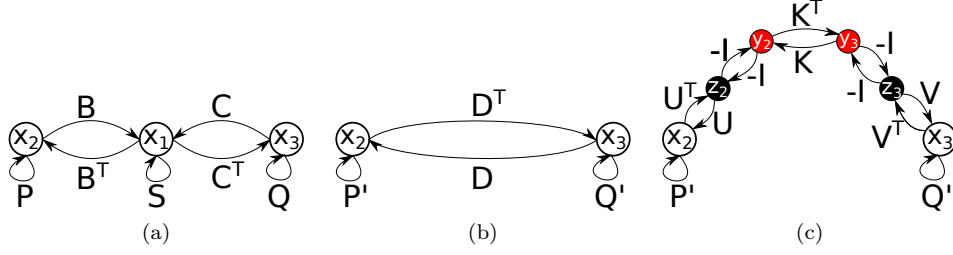


FIG. 2. An example of compression of a well-separated interaction. (a) The adjacency graph of the original linear system described in eq. (2.1). (b) The resulting graph after eliminating x_1 node. (c) The adjacency graph of the extended system. All edges are labeled with their corresponding block in the matrix.

In fig. 2a the adjacency graph of the system of equations in eq. (2.1) is shown. Now, consider eliminating x_1 . The resulting system is as follows

$$(2.2) \quad \begin{pmatrix} P' & D \\ D^T & Q' \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b'_2 \\ b'_3 \end{pmatrix},$$

where $D = -B^T S^{-1} C$, $P' = P - B^T S^{-1} B$, $Q' = Q - C^T S^{-1} C$, $b'_2 = b_2 - B^T S^{-1} b_1$, and $b'_3 = b_3 - C^T S^{-1} b_1$. The adjacency graph of the system of equations in eq. (2.2) is depicted in fig. 2b. Nodes 2 and 3 can be considered well-separated because they are not connected in the initial graph. They get connected due to the elimination of node 1. Therefore, we assume their interaction is low-rank, and can be written as below

$$(2.3) \quad D \simeq U K V^T,$$

where U and V are tall matrices. We can use any low-rank approximation in eq. (2.3), for example, singular value decomposition (SVD). We combine eqs. (2.2) and (2.3) to define a new set of equations, in which direct interaction of the nodes 2 and 3 is replaced by a sequence of low-rank interactions

$$(2.4) \quad \begin{pmatrix} P' & & U & & & \\ & Q' & & V & & \\ U^T & & & & -I & \\ & V^T & & & & -I \\ & & -I & & & K \\ & & & -I & K^T & \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \\ z_2 \\ z_3 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} b'_2 \\ b'_3 \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix}.$$

In the above equation, we introduced extra variables (y_2, y_3, z_2 , and z_3) to represent the far-field interactions. This technique is known as extended sparsification [14]. Note that the system of equations eq. (2.4) is equivalent to the system of equations eq. (2.2) up to the accuracy of eq. (2.3). In fact, if we do a Gaussian elimination on the matrix in eq. (2.4) using 2×2 blocks starting from the right, we get back to the original matrix in eq. (2.2).

In an analogy with the fast multipole method, y_2 and y_3 can be considered as the multipole coefficients, and z_2 and z_3 as the local coefficients. In fig. 2c the adjacency graph of the extended linear system is depicted. The red and black nodes correspond to the auxiliary variables introduced above. Similar to the fast multipole method,

in section 3, we employ the above technique hierarchically in a multilevel fashion to develop a sparse matrix inversion algorithm with linear complexity. As we eliminate nodes at one level, we create new nodes (e.g., y_2 and z_2) at the parent level. When all nodes at one level are eliminated, we continue the elimination at the parent level.

3. Hierarchical representation. A direct solve of $A\mathbf{x} = \mathbf{b}$ consists of two parts: factorization and solve. In the factorization part, we perform a block Gauss elimination. However, the standard block elimination process gives rise to generation of many new non-zero blocks (fill-ins) corresponding to the Schur complement of the eliminated variables. In order to achieve a factorization with linear complexity, in the proposed algorithm we compress fill-ins corresponding to the well-separated interactions. The compression process involves introducing some extra variables (e.g., y_2, z_2, y_3 , and y_3 in the example in section 2.3). Therefore, the size of the matrix during the elimination process increases while the sparsity pattern is preserved. Consequently, new nodes are added to the adjacency graph of the system of equations.

As we perform the elimination, well-separated fill-ins are created. Hence, we need to compress such edges. Similar to the example mentioned in section 2.3, for each cluster of variables we reserve a black and a red node to be used for the compression process. We consider the set of all red-nodes (corresponding to the auxiliary variables) to form the parent level. Therefore, as we proceed with the elimination process at level i , edges between the auxiliary nodes at level $i - 1$ are created. After the elimination process at level i is completed, we proceed with the elimination at level $i - 1$, and continue up to the root.

Similar to the agglomeration in multi-grid methods, we consider one red and one black node for every pair of clusters (i.e., the well-separated interactions of each pair of clusters are compressed together). Therefore, the number of red-nodes at the parent level is half of the number of clusters at the current level. In this section, we define tree structures to keep track of the auxiliary variables defined for every level.

3.1. Definitions. To form a hierarchical tree, we recursively partition the matrix. This can be formally defined as a sequence of nested partitionings that naturally implies a binary tree T . We call T the *bisection tree* of the matrix A .

DEFINITION 3.1. (*nested partitionings*) For a given matrix consider a sequence of partitionings $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_l$, such that \mathcal{P}_i maps columns/rows of the matrix to 2^i clusters (blocks). Partitionings are nested when clusters implied by \mathcal{P}_{i+1} can be constructed by dividing every cluster implied by \mathcal{P}_i into two parts.² Therefore, each cluster implied by \mathcal{P}_i is the parent of two clusters implied by the partitioning \mathcal{P}_{i+1} . An example of a nested partitioning with six levels (i.e., $\mathcal{P}_0, \dots, \mathcal{P}_5$) is illustrated in appendix B.

DEFINITION 3.2. (*bisection tree of a sparse matrix*) For a given sparse matrix A , a sequence of $l+1$ nested partitionings implies a bisection tree T with depth l . The root of the tree corresponds to the matrix A . Nodes at level i of T correspond to rows and columns of the diagonal blocks implied by \mathcal{P}_i . Since the partitionings are nested, nodes at level $i+1$ can be constructed by sub-dividing nodes in level i into two children nodes.

Now, we define the hierarchical tree³ (denoted by \mathcal{H} -tree) of a sparse matrix A

²The binary subdivision is not necessary. We can generalize this to quad-tree, octree, etc.

³In fact, it is not a tree. As we see later, there are edges between nodes at each level.

given a sequence of nested partitionings. The hierarchical tree is similar to the bisection tree; however, it consists of two types of nodes: red-nodes and black-nodes.

DEFINITION 3.3. (*hierarchical tree of a matrix*) For a given matrix A , and a nested sequence of partitionings $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_l$, the hierarchical tree is defined as follows: each node is either red or black. The root of the tree is a red-node. Every non-leaf red-node has exactly one black-child (i.e., its child is a black-node). Every black-node has exactly two red-children. There are 2^i red-nodes at level i for $i = 0, 1, \dots, l$ that are in one-to-one correspondence with level i nodes in the bisection tree implied by \mathcal{P}_i .

An example of an \mathcal{H} -tree is depicted in fig. 3. A parent-child relationship between two nodes is shown by a dashed line, whereas the interaction between two nodes (corresponding to a block in the matrix) is shown by a solid edge. We define a level for each node in the \mathcal{H} -tree. The level of the red-node at the root is 0. The level of each black-node is the level of its red parent plus 1, and the level of every non-root red-node is equal to the level of its black-parent. This is also demonstrated in fig. 3.

Each node in the \mathcal{H} -tree represents a set of variables and equations, and each edge represents a non-zero block in the matrix. Note that the tree structure is merely used to organize the variables (including the original and auxiliary variables). At every step of the algorithm, the whole tree should be considered as the adjacency graph of the extended system of equations. The non-zero blocks of the given sparse matrix with partitioning \mathcal{P}_l are represented by the edges at the leaf level of the \mathcal{H} -tree. The non-leaf nodes (shown transparent in fig. 3) of the \mathcal{H} -tree have no interaction (edge) initially, and are reserved to be used for compression similar to the example in section 2.3. The details of the algorithm are explained in section 4.

We use the following notation in the rest of the paper to denote different nodes in the \mathcal{H} -tree:

- $b_j^{[i]}$ is the j^{th} black-node at level $i \geq 1$, for $1 \leq j \leq 2^{i-1}$.⁴
- A sibling pair of red-nodes $r_{0_j}^{[i]}$ and $r_{1_j}^{[i]}$ are the j^{th} pair of red-nodes at level i , which are the left-child and right-child of their parent $b_j^{[i]}$, respectively.
- $s_j^{[i]}$ is the j^{th} super-node at level i , which consists of a pair of red-nodes $r_{0_j}^{[i]}$ and $r_{1_j}^{[i]}$. It is formed by concatenating variables and equations of $r_{0_j}^{[i]}$ and $r_{1_j}^{[i]}$. The process of merging red-nodes to a super-node is explained in section 4. $s_j^{[i]}$ is the child of $b_j^{[i]}$.
- \mathbb{P} is used to denote the parent of a node. For example, $\mathbb{P}(s_j^{[i]}) = b_j^{[i]}$.

DEFINITION 3.4. (*correspondence between the bisection tree and the \mathcal{H} -tree.*) The bisection map, \mathcal{M}_b , naturally maps each node of the \mathcal{H} -tree to a node in the bisection tree. There are 2^i red-nodes at level i of the \mathcal{H} -tree, corresponding to the partitioning \mathcal{P}_i . Hence, each red-node $r_{0_j}^{[i]}$ (similarly $r_{1_j}^{[i]}$) corresponds to a unique node $\mathcal{M}_b(r_{0_j}^{[i]})$ in level i of the bisection tree. We define $\mathcal{M}_b(b_j^{[i+1]}) = \mathcal{M}_b(s_j^{[i+1]}) = \mathcal{M}_b(\mathbb{P}(b_j^{[i+1]}))$. Note that $\mathbb{P}(b_j^{[i+1]})$ is a red-node at level i . Therefore, a non-leaf red-node, its black child node, and the super-node below it are all mapped to the same node in the bisection tree.

⁴Do not confuse $b_j^{[i]}$ with the right hand side vector \mathbf{b} in eq. (1.1).

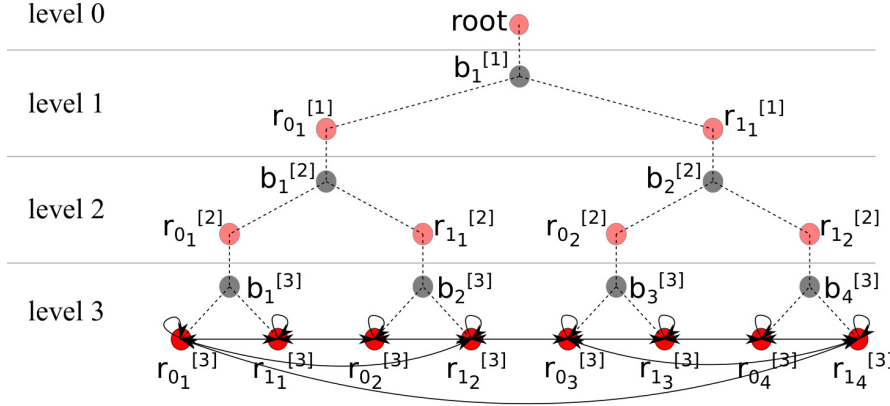


FIG. 3. An example of a hierarchical tree. Dashed lines show the parent-child relationships, while solid lines are edges corresponding to sub-blocks of the matrix. The graph induced from the leaf level is the adjacency graph of the original sparse matrix (not the extended system) with the finest partitioning. Non-leaf nodes (shown transparent) have no interaction initially, and are reserved to represent the well-separated interactions at level below them.

DEFINITION 3.5. (*adjacent nodes in a bisection tree*) Nodes p and q in a bisection tree are adjacent (or neighbors) iff at least a descendent of p is adjacent to a descendent of q . Leaf nodes are adjacent iff they are connected in the adjacency graph of the matrix with the finest partitioning \mathcal{P}_l , i.e., if the corresponding block in the matrix is non-zero.

DEFINITION 3.6. (*well-separated nodes in the \mathcal{H} -tree*) Nodes p and q in an \mathcal{H} -tree are well-separated if $\mathcal{M}_b(p)$ and $\mathcal{M}_b(q)$ are not adjacent in the bisection tree. An edge that connects two well-separated nodes is called a well-separated edge.

If the variables of the linear system correspond to points in the physical space, for example when solving a discretized PDE, two clusters of points are well-separated if the distance between the clusters is large relative to the diameters of the clusters. This is similar to the concept of well-separated clusters in the fast multipole method [49]. For a general sparse matrix, however, there is no physical information available. Thus, the concept of distance between clusters is replaced with “distance” in the adjacency graph.

4. Algorithm. The algorithm presented in this paper takes advantage of similar technique as in the inverse fast multipole method (IFMM) [19]. The IFMM can be used to compute the inverse of a dense linear system which is in the form of a FMM matrix. All levels of the hierarchical tree in the case of the IFMM are initially populated, since the \mathcal{H} -tree represents a dense matrix. However, in the case of a sparse matrix, initially there are only edges at the leaf level of the hierarchical tree representing the local interactions (corresponding to the adjacency graph of the given sparse system of equations with the partitioning \mathcal{P}_l). The tree is filled as we proceed with the elimination process.

In algorithm 1 the overall factorization scheme is introduced. Then, the algorithm is repeated for all levels. Various sub-algorithms are explained afterwards. In addition, a step by step example of the factorization process on the \mathcal{H} -tree and the corresponding

extended matrix is presented in appendix A. The factorization is followed by the solve process, which involves a forward and a backward traverse through the nodes.

Algorithm 1: Factorization using \mathcal{H} -tree.

```

Initialize( $l$ )
for  $i \leftarrow l$  to 2 do
  for  $j \leftarrow 1$  to  $2^{i-1}$  do
     $s_j^{[i]} \leftarrow \text{MergeRedNodes}(r_{0j}^{[i]}, r_{1j}^{[i]})$ 
  for  $j \leftarrow 1$  to  $2^{i-1}$  do
    Compress( $s_j^{[i]}$ )
    Eliminate( $s_j^{[i]}$ )
    Eliminate( $b_j^{[i]}$ )

```

4.1. Initializing the \mathcal{H} -tree. The `Initialize(l)` function in algorithm 1 consists of creating the \mathcal{H} -tree with depth l , and forming edges at the leaf level using non-zero blocks of the matrix. An example of the initial \mathcal{H} -tree and the corresponding matrix is depicted in fig. 20. Furthermore, in appendix B an example of a nested partitioning for a sparse matrix is shown. Variables associated to the leaf nodes are segments of the main unknown variable vector \mathbf{x} in $A\mathbf{x} = \mathbf{b}$. Furthermore, the right hand side of the leaf nodes are segments of the main right hand side vector \mathbf{b} in $A\mathbf{x} = \mathbf{b}$. The segmentation is implied by the finest partitioning of the \mathcal{H} -tree, that is \mathcal{P}_l . Variables and equations associated to the non-leaf nodes of the tree are defined during the compression process.

4.2. Creating super-nodes. The outer-loop in algorithm 1 is over different levels from the bottom to the top of the tree. At each level, we start by merging red-siblings into super-nodes. In algorithm 1 this process is denoted by the function `MergeRedNodes()`. A super-node corresponds to the concatenation of variable vectors of its constituting red-nodes, i.e.,

$$(4.1) \quad \text{var}(s_j^{[i]}) = \text{concatenate} \left(\text{var}(r_{0j}^{[i]}), \text{var}(r_{1j}^{[i]}) \right)$$

Similarly, interactions (i.e., edges) of a super-node are obtained by concatenating the interactions of its two constituting red-nodes. Therefore, the right hand side of a super-node is defined as follows:

$$(4.2) \quad \text{rhs}(s_j^{[i]}) = \text{concatenate} \left(\text{rhs}(r_{0j}^{[i]}), \text{rhs}(r_{1j}^{[i]}) \right)$$

The process of merging red-nodes to form the super-nodes is illustrated in fig. 4. The merging process is also depicted in fig. 21 in the example provided in appendix A. After creating the super-nodes, we go over every super-node in the current level, compress its well-separated interactions, eliminate it (i.e., its two constituting red-nodes), and finally eliminate its black-parent.

4.3. Compressing well-separated edges. The next sub-algorithm to consider is the *compression*. In algorithm 1 this process is denoted by the function `Compress()`. During the compression, well-separated interactions of a super-node are pushed to the

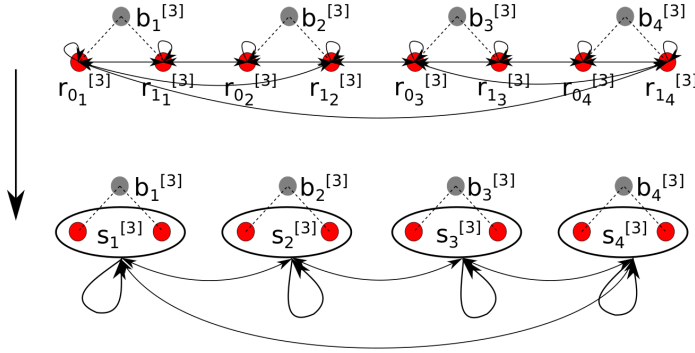


FIG. 4. An example of the merge process corresponding to the `MergeRedNodes()` function in algorithm 1. The red-nodes of the \mathcal{H} -tree in fig. 3 (top) are merged to super-nodes (bottom).

parent level nodes which represent a set of auxiliary variables. This is essentially the extended sparsification method which we discussed in the example in section 2.3 (i.e., replacing the well-separated edges in fig. 2b by the sequence of edges between red and black nodes at the parent level as shown in fig. 2c).

Assume we are at level i , and about to apply `Compress`($s_j^{[i]}$). Also, assume $s_j^{[i]}$ is of size m (i.e., corresponds to m variables and m equations), and interacts with (i.e., has an edge to) t well-separated nodes p_1, p_2, \dots, p_t with sizes m_1, m_2, \dots, m_t , respectively. Node p_k for $k = 1, \dots, t$ can be a red-node (at the parent level) or a super-node (at the same level). Assume blocks A_1, A_2, \dots, A_t are associated to the outgoing well-separated edges from $s_j^{[i]}$ to p_1, p_2, \dots, p_t , respectively, where A_k is an m_k by m block. Similarly, $B_1^T, B_2^T, \dots, B_t^T$ are associated to the incoming well-separated edges to $s_j^{[i]}$, where B_k is an m_k by m block. This is depicted schematically in fig. 5.

Similar to the example in section 2.3, we assume well-separated edges can be approximated using a low-rank factorization. We compress all well-separated interactions of a super-node together. We then introduce auxiliary variables, and replace the well-separated edges by new edges between the auxiliary variables (i.e., going from the left configuration to the right configuration in fig. 5). Note that the new system of equations is equivalent to the original system up to the accuracy of the low-rank approximation used.

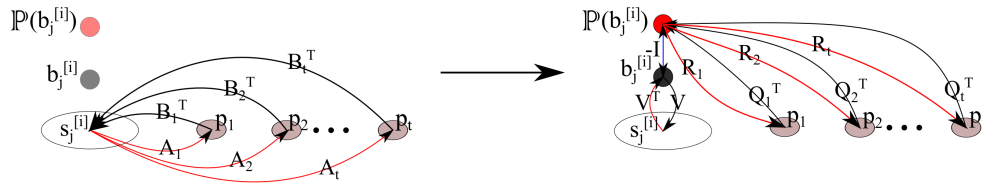


FIG. 5. Schematic of the compression process for a super-node $s_j^{[i]}$ corresponding to the `Compress`() function in algorithm 1. Well-separated interactions are replaced with low-rank interactions with the red parent.

In order to compress all t well-separated edges together, we form a temporary matrix by vertically concatenating A_1, \dots, A_t as well as B_1, \dots, B_t . Use a low-rank

approximation method (e.g., SVD) and write:

$$(4.3) \quad \begin{bmatrix} A_1 \\ \vdots \\ A_t \\ B_1 \\ \vdots \\ B_t \end{bmatrix} \simeq \begin{bmatrix} R_1 \\ \vdots \\ R_t \\ Q_1 \\ \vdots \\ Q_t \end{bmatrix} V^\top,$$

where R_k and Q_k are m_k by r matrices for $k = 1, 2, \dots, t$, and V is an m by r matrix. r is the rank in the above low-rank approximation. From eq. (4.3) we can write:

$$(4.4) \quad A_k \simeq R_k V^\top \quad \text{for } k = 1, 2, \dots, t.$$

Variables of the super-node $s_j^{[i]}$ contribute to the equations (matrix rows) associated to the node p_k with a term like $A_k \mathbf{var}(s_j^{[i]})$ for $k = 1, 2, \dots, t$. Therefore, this contribution can be written as:

$$(4.5) \quad A_k \mathbf{var}(s_j^{[i]}) \simeq (R_k V^\top) \mathbf{var}(s_j^{[i]}) = R_k (V^\top \mathbf{var}(s_j^{[i]})).$$

The term $V^\top \mathbf{var}(s_j^{[i]})$ is a new vector of variables with r unknowns (similar to the vector y_2 in the example mentioned in section 2.3). We assign this vector of auxiliary variables to the red-node $\mathbb{P}(b_j^{[i]})$. Therefore, we have

$$(4.6) \quad V^\top \mathbf{var}(s_j^{[i]}) - \mathbf{var}(\mathbb{P}(b_j^{[i]})) = \mathbf{0}.$$

We associate equation eq. (4.6) to the black-node $b_j^{[i]}$. Hence, $\mathbf{rhs}(b_j^{[i]}) = \mathbf{0}$. Based on eq. (4.6), there are edges from $s_j^{[i]}$ and $\mathbb{P}(b_j^{[i]})$ to $b_j^{[i]}$ with blocks V^\top and $-\mathbb{I}_r$ (minus identity matrix of size r), respectively. Also, every outgoing edge from $s_j^{[i]}$ to a well-separated node p_k is substituted by an outgoing edge from $\mathbb{P}(b_j^{[i]})$ to p_k associated with the block R_k . This is illustrated in fig. 5.

Note that based on eq. (4.5) $A_k \mathbf{var}(s_j^{[i]}) \simeq R_k \mathbf{var}(\mathbb{P}(b_j^{[i]}))$. Hence, introducing the auxiliary variables $\mathbf{var}(\mathbb{P}(b_j^{[i]}))$ results in an equivalent system of equations up to the accuracy of the low-rank approximation in eq. (4.3).

Similarly, we perform the compression process for the incoming well-separated edges to the supernode $s_j^{[i]}$. From eq. (4.3) we have

$$(4.7) \quad B_k \simeq Q_k V^\top \Rightarrow B_k^\top = V Q_k^\top \quad \text{for } k = 1, 2, \dots, t.$$

Therefore, the net contribution of the variables associated to p_1, \dots, p_t in the equation associated to the super-node $s_j^{[i]}$ can be written as follows

$$(4.8) \quad \sum_{k=1}^t B_k^\top \mathbf{var}(p_k) \simeq \sum_{k=1}^t (V Q_k^\top) \mathbf{var}(p_k) = V \sum_{k=1}^t Q_k^\top \mathbf{var}(p_k).$$

The term $\sum_{k=1}^t Q_k^\top \mathbf{var}(p_k)$ is a new vector of variables with r unknowns (similar to the vector z_2 in the example mentioned in section 2.3). We assign this vector of

auxiliary variables to the black-node $b_j^{[i]}$. Therefore,

$$(4.9) \quad \sum_{k=1}^t Q_k^T \mathbf{var}(p_k) - \mathbf{var}(b_j^{[i]}) = \mathbf{0}.$$

eq. (4.9) is assigned to the red-node $\mathbb{P}(b_j^{[i]})$. Hence, based on eq. (4.9), there are edges from $b_j^{[i]}$, and p_k for $k = 1, \dots, t$ to $\mathbb{P}(b_j^{[i]})$ with blocks $-\mathbb{I}_r$, and Q_k^T for $k = 1, \dots, t$, respectively. Additionally, based on eq. (4.8) and eq. (4.9), the net contribution of all well-separated nodes p_1, p_2, \dots, p_t to the super-node $s_j^{[i]}$ is replaced by a single term $V \mathbf{var}(b_j^{[i]})$. This adds an edge from $b_j^{[i]}$ to $s_j^{[i]}$ with block V . This is illustrated in fig. 5.

Therefore, in the compression process all well-separated edges connected to the super-node $s_j^{[i]}$ are substituted with edges to/from $\mathbb{P}(b_j^{[i]})$, as shown in fig. 5. As a result of the compression process on a super-node $s_j^{[i]}$, we defined $2r$ new auxiliary variables as well as $2r$ new equations. These new variables and equations are associated to $b_j^{[i]}$ and $\mathbb{P}(b_j^{[i]})$ as explained above. $\mathbf{var}(b_j^{[i]})$ and $\mathbf{var}(\mathbb{P}(b_j^{[i]}))$, respectively, are the general form of the auxiliary variables z_2 and y_2 in the example of section 2.3. In the example provided in appendix A, figs. 23 and 26 illustrate the graph and matrix after the compression is applied.

Note that if the matrix is symmetric, $A_k = B_k$ for $k = 1, \dots, t$. Therefore, we would not need to concatenate B_k 's in eq. (4.3), and only half of the above calculations are required.

4.4. Elimination. After compressing all well-separated edges, we apply the standard elimination. As a result of the compression process, the super-node $s_j^{[i]}$ is only connected to its neighbor nodes. This is a key property of the algorithm that preserves the sparsity pattern of the matrix. The elimination process for a node is explained in section 2.2. We first eliminate the super-node $s_j^{[i]}$, and then eliminate its black-parent $b_j^{[i]}$. In the example provided in appendix A, figs. 22, 24, 25 and 27 illustrate the graph and matrix after the elimination process.

4.5. Solve. After the factorization part is completed, we can solve for multiple right hand sides. The solve process consists of two steps: a forward and a backward traversal of all nodes. This is identical to the standard forward and backward substitutions in the LU factorization. In the forward traversal we visit all nodes in the order they have been eliminated, and in the backward traversal we visit nodes in the exact reverse order. The solve process is introduced in algorithm 2.

Note that in the factorization part we introduced auxiliary variables and equations (i.e., all variables and equations associated to non-leaf nodes). We denote this *extended* system of equations by $A_e \mathbf{x}_e = \mathbf{b}_e$, where the vectors \mathbf{x} and \mathbf{b} in eq. (1.1) (corresponding to the leaf super-nodes) are part of the vectors \mathbf{x}_e and \mathbf{b}_e , respectively. In the solve part, we have to solve for all variables (original and auxiliary variables, i.e., \mathbf{x}_e) even though we are just interested in the original variables, \mathbf{x} . The number of extra variables is limited, and is of the same order as the number of the original variables.

For a given right hand side \mathbf{b} , the solve algorithm begins with setting the right hand side for every leaf node in the tree. The right hand side of each leaf-node is a segment of the vector \mathbf{b} determined by the leaf-partitioning of the \mathcal{H} -tree, i.e., \mathcal{P}_l .

Algorithm 2: Solve for a given right hand side using \mathcal{H} -tree.

```

SetRHS()
for  $i \leftarrow l$  to 2 do
    for  $j \leftarrow 1$  to  $2^{i-1}$  do
        SolveL( $s_j^{[i]}$ )
        SolveL( $b_j^{[i]}$ )
    for  $i \leftarrow 2$  to  $l$  do
        for  $j \leftarrow 2^{i-1}$  to 1 do
            SolveU( $b_j^{[i]}$ )
            SolveU( $s_j^{[i]}$ )
            SplitVar( $s_j^{[i]}$ )

```

The right hand side of the super-nodes are formed by concatenating the right hand side of their two constituting red-nodes as explained in eq. (4.2). The right hand side of all other nodes in the tree is initially set to $\mathbf{0}$ as explained in section 4.3.

The rest of the solve algorithm consists of two functions `solveL()` and `solveU()` that are applied to all super-nodes and black-nodes. These functions are explained in algorithms 3 and 4. After solving for variables of a super-node, we split the solution between its two constituting red-nodes via the function `SplitVar()` according to eq. (4.1).

Algorithm 3: Forward traversal (update the right hand side of all nodes)

```

Function SolveL(node  $p$ )
     $f \leftarrow \text{Pivot}(p)^{-1} \cdot \text{RHS}(p)$ 
    for  $e \in \text{OutGoingEdges}(p)$  do
        if  $\text{Order}(e.\text{head}) > \text{Order}(p)$  then
             $\text{RHS}(e.\text{head}) \leftarrow \text{RHS}(e.\text{head}) - e.\text{block} \cdot f$ 

```

Algorithm 4: Backward traversal (solve for variables of all nodes)

```

Function SolveU(node  $p$ )
     $\text{Var}(p) \leftarrow \text{RHS}(p)$ 
    for  $e \in \text{InComingEdges}(p)$  do
        if  $\text{Order}(e.\text{tail}) > \text{Order}(p)$  then
             $\text{Var}(p) \leftarrow \text{Var}(p) - e.\text{block} \cdot \text{Var}(e.\text{tail})$ 
     $\text{Var}(p) \leftarrow \text{Pivot}(p)^{-1} \cdot \text{Var}(p)$ 

```

In algorithms 3 and 4 `OutGoingEdges(p)` and `InComingEdges(p)` denote the set of all outgoing and incoming edges of a node p , respectively. During the factorization process, the blocks associated to edges connecting to a node can get updated until

that node is eliminated. Essentially, these updated edges are the outcome of the factorization part. The solve process in the proposed algorithm is then identical to the standard forward and backward substitutions. The only difference with the standard block LU factorization occurs during the factorization step, in which we replace well-separated edges with a sequence of edges between auxiliary variables.

An edge e is assumed to connect the node $e.\text{tail}$ (the source node) to the node $e.\text{head}$ (the destination node). $e.\text{block}$ denotes the block associated to the edge e , i.e., the variables associated to the node $e.\text{tail}$ contribute to the equations associated to the node $e.\text{head}$ via a term like $e.\text{block} \cdot \text{Var}(e.\text{tail})$. In addition, $\text{Pivot}(p)$ refers to the matrix block corresponding to the self-edge (i.e., the edge from node p to itself) of node p . The function $\text{Order}(p)$ returns the order of elimination of different nodes, that is if in the factorization part a node q is eliminated before a node p , then $\text{Order}(q) < \text{Order}(p)$.

Algorithm 3 performs the forward traversal with the same order as the elimination, and updates the right hand side vector of each node. Algorithm 4 performs the backward traversal, and solves for variables of each node. The original vector of variables \mathbf{x} , which is part of the extended vector \mathbf{x}_e , is obtained after this step.

5. Linear complexity. In this section we show that the block sparsity pattern of the extended matrix is preserved through the elimination process. Therefore, the factorization algorithm has provable linear complexity provided that the block sizes (and thus the rank of the low-rank approximations) are bounded.

In section 3.1 we defined the bisection-tree based on a sequence of nested partitionings, $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_l$. We call the adjacency graph of the matrix A with partitioning \mathcal{P}_i the adjacency graph of level i .

We define the distance between two nodes p and q respectively at levels i and j of the bisection tree as follows. Assume $i \leq j$, and node w is the ancestor of q at level i of the bisection tree. Therefore, p and w are vertices of the adjacency graph of level i . The distance between p and q is defined as the length of (i.e., number of edges) the minimum path between p and w in the adjacency graph of level i .

Furthermore, the distance between two nodes $p_{\mathcal{H}}$ and $q_{\mathcal{H}}$ in the \mathcal{H} -tree (they can be red, black, or super nodes) is defined as the distance between $\mathcal{M}_b(p_{\mathcal{H}})$ and $\mathcal{M}_b(q_{\mathcal{H}})$ (see definition 3.4).

THEOREM 5.1. (*preservation of sparsity pattern*) *In the \mathcal{H} -tree factorization algorithm, we never create an edge between two nodes with distance greater than 2.*

Proof. We prove the theorem using induction on the number of eliminated super-nodes in the tree.

Induction hypothesis: Assume we have eliminated k super-nodes so far, and we are about to eliminate the $(k+1)^{\text{th}}$ super-node. The induction hypothesis states that there is no edge between nodes with distance greater than 2. We need to show that eliminating the next super-node (i.e., the $(k+1)^{\text{th}}$ super-node) and its black-parent does not result in creating such an edge.

The induction basis is for $k = 0$, i.e., when no super-node is eliminated. In this case, by construction, we are at the leaf level, and all super-nodes are connected to other super-nodes within distance 0 or 1. Hence, eliminating a super-node only results in the creation of edges between neighbors of neighboring super-nodes (i.e., at most distance 2). For $k = 0$ the black-parent is disconnected from the graph (the first super-node has no well-separated edges to be compressed). Therefore, elimination of

the black-parent does not create any new edge.

The induction step: From the induction hypothesis the $(k+1)^{\text{th}}$ super-node has edges to other nodes (either a super-node at the same level or a red-node at the parent level) with distance at most 2. Nodes that are with distance 2 from each other, are well-separated (see definitions 3.4, 3.5, and 3.6). Based on algorithm 1, we compress any edge from the $(k+1)^{\text{th}}$ super-node to other nodes with distance 2 before elimination. Hence, at the time of elimination the $(k+1)^{\text{th}}$ super-node has edges to other nodes with distances less or equal to 1. Consequently, new edges created after elimination only connect super-nodes that are neighbors of a neighbor. Note that the $(k+1)^{\text{th}}$ super-node and its black-parent have the same connectivity at the elimination step. Hence, eliminating the black-parent only results in updating edges between super-nodes that are neighbors of a neighbor.

Note that during the compression process, a distance 2 edge (i.e., an edge that connects two nodes with distance 2) between two super-nodes at level i is eventually represented as a distance 2 edge between two red-nodes at level $i-1$. At the beginning of the elimination process for level $i-1$ we merge the red-nodes and create super-nodes (see section 4.2). Since all edges between red-nodes are at most distance 2 edges, the edges between the super-nodes are also at most distance 2. Hence, the induction argument is still valid when going from level i to $i-1$. \square

theorem 5.1 guarantees that during the factorization phase for each node we need to process at most $\kappa_1 + \kappa_2$ edges. For a given super-node, κ_1 is the maximum number of adjacent super-nodes, and κ_2 is the maximum number of super-nodes at distance 2 in the original adjacency graph. Note that κ_1 and κ_2 depend on the original matrix sparsity pattern, and are independent of the size of the matrix. To establish linear complexity of the factorization, we need to bound the size of the nodes in the \mathcal{H} -tree (and therefore, the size of blocks associated to the edges).

For matrices arising from the discretization of a PDE, well separated edges correspond to the interaction of points that are physically far from each other. Therefore, if the Green's function of the associated PDE is smooth enough, one can expect the well-separated interaction to be numerically low-rank. We provide numerical evidence in section 6 to support this argument.

For general sparse matrices we can guarantee the linear complexity through bounding the rank growth. This is explained in the next theorem.

THEOREM 5.2. *(condition for linear complexity) Consider p_i to be the maximum size of a super-node at level i of an \mathcal{H} -tree with l levels. Also, assume that p_i is bounded by a geometric series,*

$$(5.1) \quad p_i < \alpha^{l-i} p_l,$$

where $\alpha < \sqrt[3]{2}$ is a constant number. Under this condition the cost of the algorithm is linear with respect to the problem size.

Proof. For a given super-node at level i the compression cost is $\mathcal{O}(\kappa_2 p_i^3)$, and the elimination cost is $\mathcal{O}(\kappa_1^2 p_i^3)$. Note that the required memory scales with p_i^2 for each node. Ignoring the constant factors κ_1 and κ_2 , the order of the total cost of factorization is as follows:

$$(5.2) \quad \text{factorization cost} = \mathcal{O}\left(\sum_{i=1}^l 2^{i-1} p_i^3\right).$$

Plug eq. (5.1) in eq. (5.2):

$$\begin{aligned}
 \text{factorization cost} &= \mathcal{O} \left(\sum_{i=1}^l 2^{i-1} \alpha^{3(l-i)} p_l^3 \right) \\
 (5.3) \qquad &= \mathcal{O} \left(2^{l-1} p_l^3 \sum_{i=0}^{l-1} \left(\frac{\alpha^3}{2} \right)^i \right) \\
 &= \mathcal{O}(2^l p_l^3)
 \end{aligned}$$

Note that for $\alpha < \sqrt[3]{2}$ we have $\sum_{i=0}^{l-1} \left(\frac{\alpha^3}{2} \right)^i = \mathcal{O}(1)$. Furthermore, p_l is the number of variables in super-nodes at leaf level which is $\mathcal{O}(n/2^l)$. Therefore:

$$(5.4) \qquad \text{factorization cost} = \mathcal{O}(np_l^2), \quad \text{factorization memory} = \mathcal{O}(np_l).$$

□

6. Numerical results. We have implemented the algorithm described in section 4 in C++. The code (we call it LoRaSp⁵) can be downloaded from bitbucket.org/hadip/lorasp. We use Eigen [29] as the backend for linear algebra calculations, and SCOTCH [48] for graph partitioning. A graphical example of partitioning using SCOTCH is provided in appendix B. We present results for various benchmarks, where LoRaSp is used as a direct solver, or as a preconditioner in conjunction with an iterative solver.

6.1. LoRaSp as a stand-alone solver. In this section we employ LoRaSp as a stand-alone solver. The accuracy of the solver depends on the accuracy of the low-rank approximations during the compression step as explained in section 4.3. Any low-rank approximation method can be used for the compression. Here, we use SVD. For every well-separated interaction, we first compute the SVD, and then truncate the singular values at some point. There are many possible criteria to truncate singular values. We discuss some possible criteria. fig. 6 shows the decay of singular values for blocks corresponding to the interaction between randomly chosen well-separated nodes at different levels of an \mathcal{H} -tree. The tree corresponds to a matrix obtained from the second-order uniform discretization of the Poisson equation:

$$\begin{aligned}
 (6.1) \qquad \nabla \cdot (\nabla T) &= f \quad \text{in } \mathcal{D}, \\
 T &= \mathbf{0} \quad \text{on } \partial\mathcal{D}.
 \end{aligned}$$

The domain \mathcal{D} is a three-dimensional unit cube. The matrix size is 32,768, and the depth of the corresponding \mathcal{H} -tree is 11. Evidently, singular values have exponential decay at different levels of the tree. The zero (up to machine precision) singular values are not shown in the plot.

To demonstrate the linear complexity of the method, we considered a sequence of problems with a growing number of variables. Consider the following sequence of uniform discretization of the domain \mathcal{D} in eq. (6.1): $32 \times 32 \times 16$, $32 \times 32 \times 32$, $64 \times 32 \times 32$, $64 \times 64 \times 32$, $64 \times 64 \times 64$, $128 \times 64 \times 64$, and $128 \times 128 \times 64$. The matrix size is increased by a factor of 2 in the consecutive problems. Hence, to keep the size of the leaf super-nodes constant among all problems, we consider \mathcal{H} -trees with

⁵Low Rank Sparse solver.

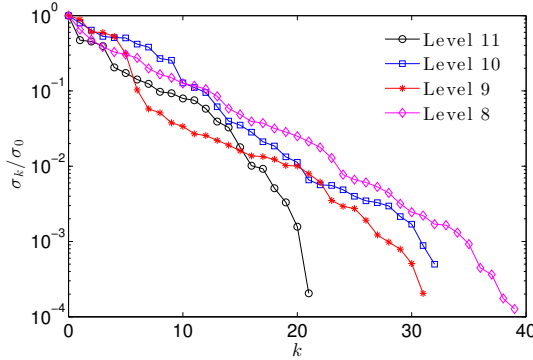


FIG. 6. Decay of singular values of random blocks corresponding to well-separated interactions at different levels of an \mathcal{H} -tree with 11 levels.

depth 10, 11, 12, 13, 14, 15, 16 for this sequence of problems, respectively. In general, the depth of \mathcal{H} -tree should scale linearly with $\log_2 n$, where n is the size of matrix.

Well-separated edges corresponding to a block B , as shown in eq. (4.3), with singular-values $\sigma_0, \sigma_1, \dots$, are compressed by keeping only the singular-values that satisfy:

$$(6.2) \quad \frac{\sigma_k}{\sigma_0} \geq \epsilon.$$

Smaller values of ϵ lead to more accurate approximation of each block, and consequently a more accurate approximation of the final solution. For a given linear system $A\mathbf{x} = \mathbf{b}$, the precision of any solution $\tilde{\mathbf{x}}$ is quantified by the relative error and relative residual defined as follows:

$$(6.3) \quad \text{error} = \frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|_2}{\|\mathbf{x}\|_2}, \quad \text{residual} = \frac{\|A\tilde{\mathbf{x}} - \mathbf{b}\|_2}{\|\mathbf{b}\|_2}.$$

fig. 7b shows that smaller values of ϵ (i.e., more accurate low-rank approximations) results in a more accurate estimation of the solution to the linear system in the cost of larger factorization and solve times, as shown in fig. 7a. For a constant ϵ , the time spent for the factorization and solve parts asymptotically scale linearly with the size of the problem. Note that for smaller values of ϵ the linear scaling is achieved for larger values of n . In addition, the error and residual of the estimated solution for a fixed ϵ barely change with the problem size.

As it is clear from fig. 7, we can obtain more accurate solutions by decreasing the parameter ϵ in eq. (6.2). To show the convergence of the solver, we picked a fixed problem size, and measured the quality of the estimated solution as ϵ decreases. In addition, for comparison purposes, we consider a 2D variation of eq. (6.1) which is discretized using a finite volume approach with Voronoi tessellation as depicted in fig. 8. The points are drawn from a random uniform distribution in the $[0, 1]^2$ interval. The discretization results in a matrix $A = DB$, where D is a diagonal matrix with inverse of the Voronoi cells on the diagonal, and B is a symmetric matrix. We apply the factorization directly on B . Note that the average number of non-zeros per row for a matrix corresponding to a 2D Voronoi discretization is 7, which is the same as for a uniform second order 3D discretization.⁶

⁶We can show this by double counting the angles in a 2D Voronoi tessellation, once through

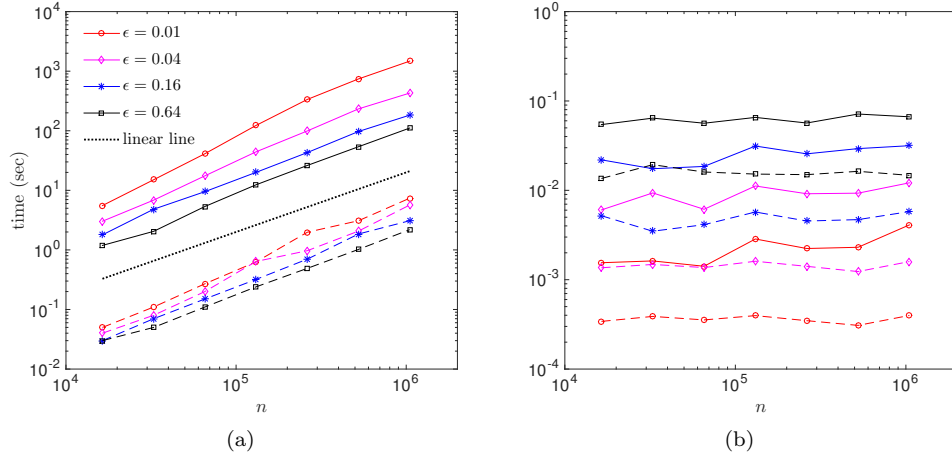


FIG. 7. Performance of the solver for different problem sizes using different levels of precision (shown by different colors and symbols) in low-rank approximations. (a) Time spent on factorization (solid line) and solve (dashed line) parts, (b) error (solid line) and residual (dashed line) of the solution.

In fig. 9 the convergence of the solution for a 3D Poisson problem with $n = 1.3 \times 10^5$ (corresponding to a $64 \times 64 \times 32$ grid) and a 2D Poisson problem with the same size (corresponding to Voronoi tessellation) are shown. The error and residual decrease proportional to the precision in the low-rank approximations, ϵ . Furthermore, it is clear that the residual is smaller than the error. The ratio of the error to residual is generally an increasing function with respect to the condition number of the matrix. For the same number of points, the condition number of a 3D discretization is lower than that of a 2D discretization. Therefore, the error and residual are closer in the 3D case in comparison to the 2D case as illustrated in fig. 9b.

fig. 9a demonstrates the factorization time as a function of the low-rank approximation precision, ϵ . The 2D and 3D cases have equal matrix size and average number of non-zeros per row; however, the factorization time is much greater in the latter. Three dimensionality of the problem leads to a higher number of well-separated interactions; hence, after every elimination more new fill-ins are introduced in the 3D case.

fig. 10 shows the breakdown of the time spent on different parts of the algorithm, namely, low-rank approximation of well-separated blocks (here, we use SVD), general matrix multiplication (gemm), and computing the inverse of pivot blocks. Clearly, for the 3D case most of the time is spent on SVD, which is known to be an expensive algorithm for low-rank approximation. This can be improved significantly if faster low-rank approximation methods are employed. We discuss some of the alternative methods in section 7. The total cost grows as ϵ decreases, similar to the results shown in fig. 9a.

In fig. 11, the average ranks of the well-separated interactions at each level are depicted as a function of the low-rank approximation precision. Note that for both the 3D and 2D cases an \mathcal{H} -tree with depth $l = 13$ is used, i.e., there are 16 variables per

points, and once through triangles of the corresponding Delaunay triangulations.

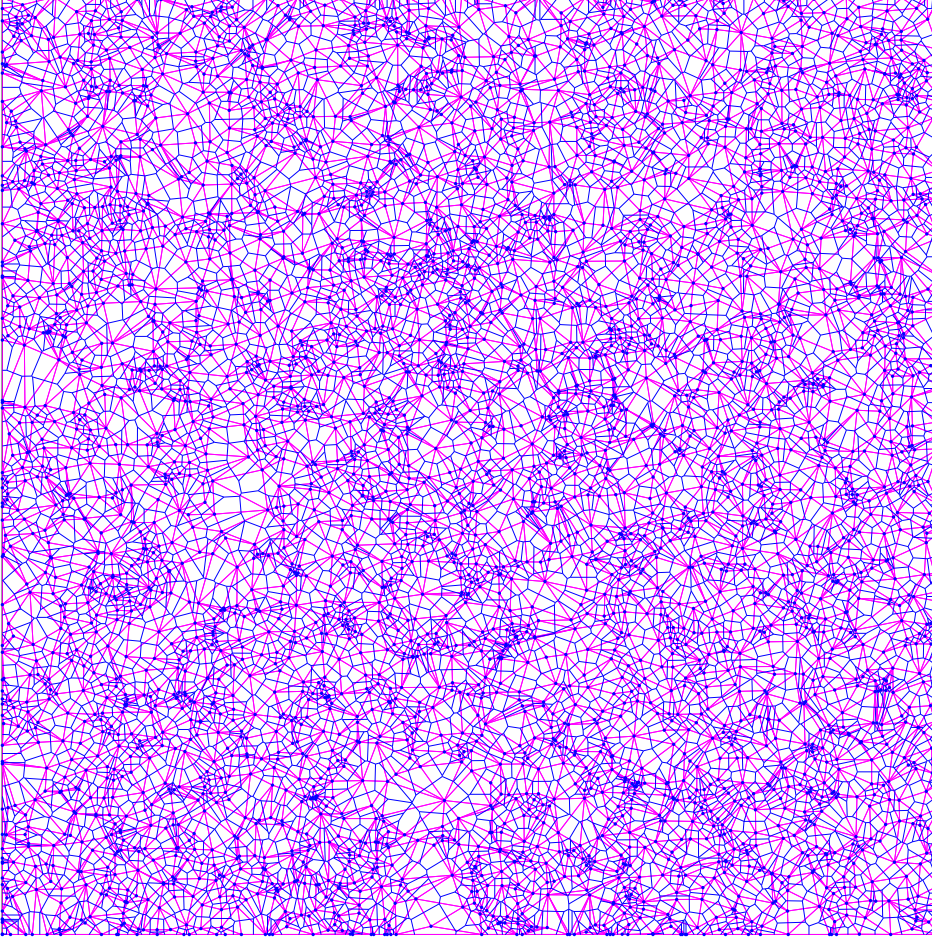


FIG. 8. An example of Voronoi tessellation of the unit square using 3,200 internal points, and 200 boundary points. The corresponding adjacency graph is also illustrated.

leaf red-nodes on average. The rank for the 3D case increases dramatically compared to the 2D case when a more accurate solution is desired. If p_L is the maximum rank among all levels (i.e., maximum size of a red-node), similar to the analysis of theorem 5.2, the factorization complexity is $\mathcal{O}(np_L^2)$. From figs. 6 and 11 we can observe that $p_L = \mathcal{O}(\log \frac{1}{\epsilon})$. Therefore, we have:

$$(6.4) \quad \text{factorization cost} = \mathcal{O}\left(n \log^2 \frac{1}{\epsilon}\right), \quad \text{factorization memory} = \mathcal{O}\left(n \log \frac{1}{\epsilon}\right).$$

6.2. LoRaSp as a preconditioner. In section 6.1 we showed that for a fixed low-rank approximation precision, and therefore solution accuracy, the total cost of the algorithm grows linearly with the problem size. However, as suggested by figs. 9 and 10 obtaining a high accuracy solution may be expensive for some problems. One standard remedy in that case is to use the low-accuracy solver as a high-accuracy preconditioner in conjunction with an iterative solver. This is particularly very appealing here, since

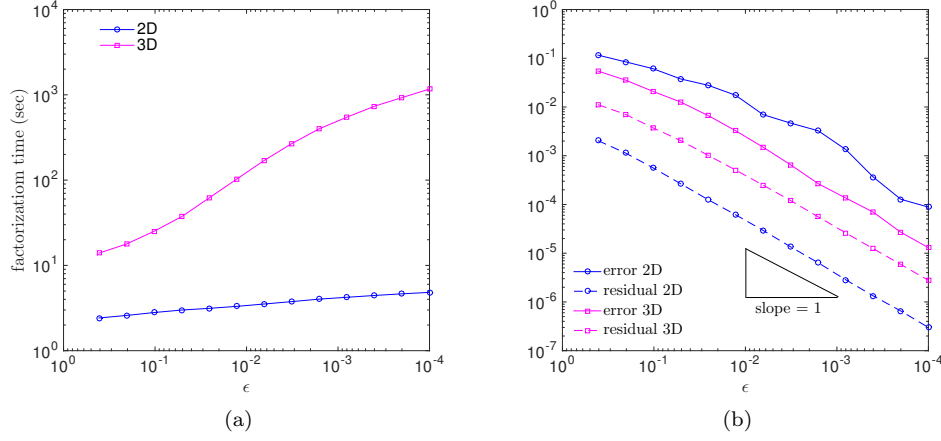


FIG. 9. (a) Factorization time as a function of the low-rank approximation precision (ϵ) for the 3D and 2D Poisson problems of size 1.3×10^5 ; (b) error and residual of the solution.

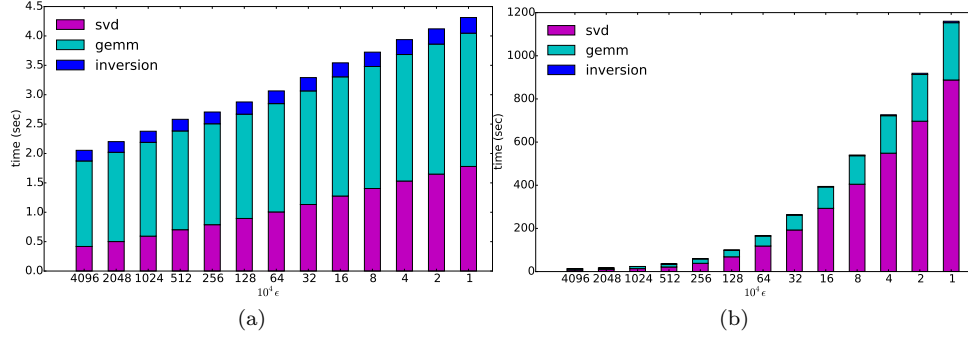


FIG. 10. Breakdown of the time spent on different parts as a function of the low-rank approximation precision for the (a) 2D and (b) 3D cases.

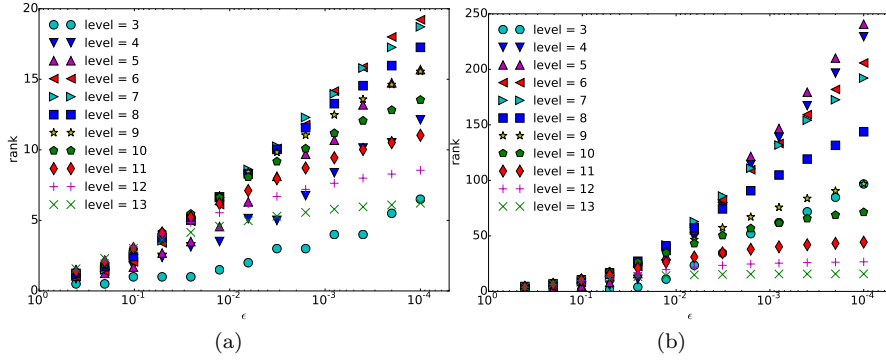


FIG. 11. Average rank of well-separated interactions per level as a function of the low-rank approximation precision for the (a) 2D and (b) 3D cases.

the factorization part is completely separated from the solve part. Therefore, we can factorize the matrix only once, and apply the (cheap) solve part at every iteration. Here, we use the GMRES method [53] as the iterative solver in conjunction with the proposed algorithm as a preconditioner for all benchmarks. Note that for matrices with specific properties such as a symmetric positive definite (SPD) matrix, one can use a more optimized iterative method (e.g., conjugate gradient in the case of SPD matrix).

6.2.1. Poisson equation (structured grid). As the first benchmark, we consider the sequence of 3D Poisson problems introduced in section 6.1. We use $\epsilon = 10^{-1}$ to factorize the matrix, and find an approximation \tilde{A} of A^{-1} . Factorization and solve times are shown in fig. 7a. We solve the system of equation $\tilde{A}A\mathbf{x} = \tilde{A}\mathbf{b}$ through GMRES afterwards. Since the solve part of the proposed algorithm is much cheaper compared to the factorization part, each GMRES iteration is also relatively cheap. In fig. 12 the sparsity pattern of the original and preconditioned matrices are shown. The preconditioned matrix, $\tilde{A}A$, approaches to the identity matrix as ϵ decreases.

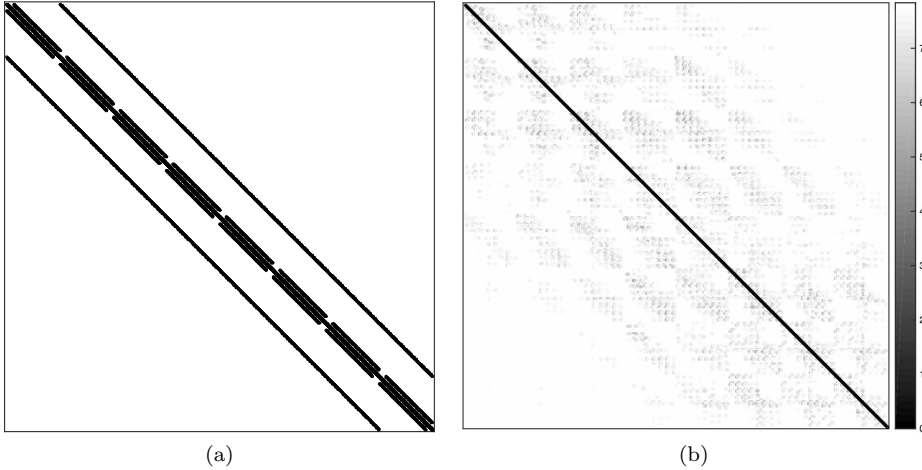


FIG. 12. (a) Sparsity pattern of a matrix A obtained from a discretization of eq. (6.1). (b) Sparsity pattern of the preconditioned matrix $\tilde{A}A$ using LoRaSp with $\epsilon = 10^{-1}$. A non-zero entry a is colored by $-\log|a|$ (i.e., larger values are darker).

In fig. 13, the GMRES residual as a function of the iteration number is plotted for different problem sizes, when LoRaSp with $\epsilon = 10^{-1}$ is used as a preconditioner. At every iteration of the preconditioned GMRES method we have an approximation $\tilde{\mathbf{x}}$ of the solution. The residual in this case is defined as follows:

$$(6.5) \quad \text{preconditioned GMRES residual} = \frac{\|\tilde{A}\mathbf{b} - \tilde{A}A\tilde{\mathbf{x}}\|_2}{\|\tilde{A}\mathbf{b}\|_2}.$$

The number of iterations that GMRES needs to converge slightly grows with size of the problem. This is due to growth of the condition number of the problem. Similar to multi-grid methods, we can use specific knowledge about the underlying PDE and discretization to obtain problem-size independent convergence. This is the topic of our future work, and is not discussed in this paper. In fig. 14 the condition number, $\kappa(A)$, is plotted as a function of the matrix size. The condition numbers are approximated

using the 1-norm [34, 37]. Note that for a matrix of size n corresponding to the second order finite difference discretization of the Poisson equation, we expect the condition number to grow as $n^{2/3}$. The $n^{2/3}$ trend is also depicted in fig. 14.

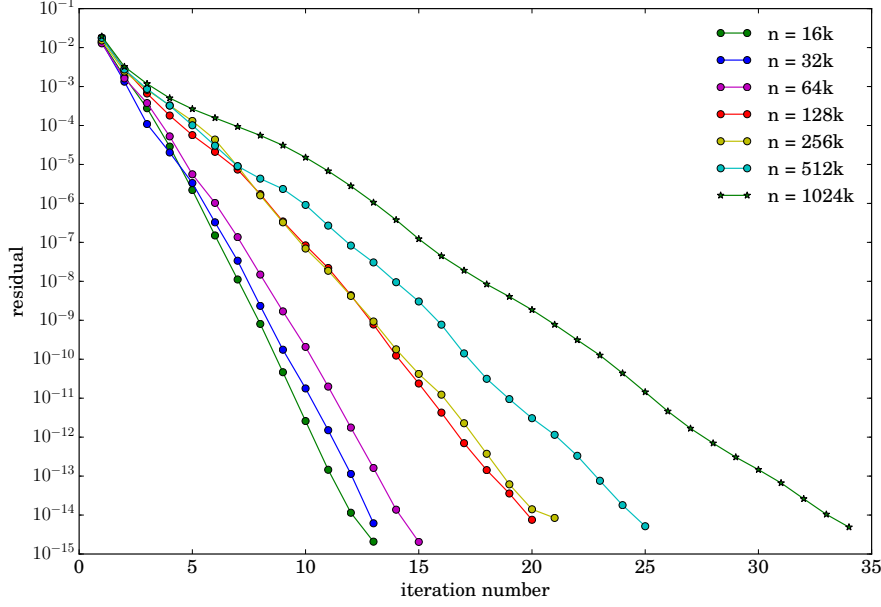


FIG. 13. *GMRES* residual as a function of the iteration number for different problem sizes. *LoRaSp* is used as a preconditioner with $\epsilon = 10^{-1}$.

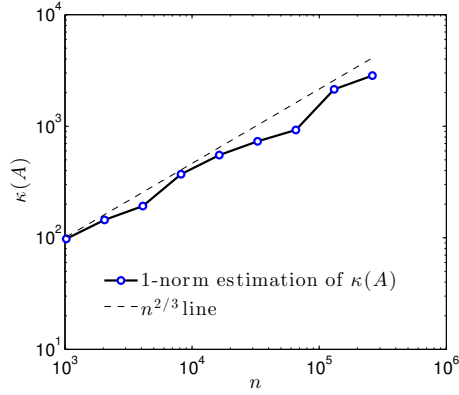


FIG. 14. Condition number versus matrix size for a sequence of matrices obtained from the discretization of eq. (6.1).

6.2.2. Variable coefficient Poisson equation (structured grid). As our next benchmark we consider the variable coefficient Poisson equation with periodic boundary conditions discretized on a three-dimensional uniform grid:

$$(6.6) \quad \nabla \cdot (\phi \nabla T) = f$$

In the above equation, the scalar fields ϕ and f are given, and we solve for T , similar to eq. (6.1). We consider three cases for the coefficient field, ϕ :

- **case 1:** At each point of the domain, ϕ is drawn from a uniform distribution, $\text{unif}(0, 1)$, independently.
- **case 2:** At each point of the domain, ρ is drawn from a uniform distribution, $\text{unif}(0, 1)$, independently. ϕ is then defined as $\phi = \frac{1}{\rho}$.
- **case 3:** At each point of the domain, ϕ is drawn from a uniform distribution, $\text{unif}(-1, 1)$, independently.

For the two first cases, the corresponding matrices are symmetric negative definite. Case 2 shows up in the numerical simulation of a variable-density flow in the low-Mach number limit [16, 30, 50], where in that case T and ρ are hydrodynamic pressure and density of the flow, respectively. The third case, however, results in an indefinite matrix.

In fig. 15a, the norm of the eigenvalues of the matrices corresponding to a 16^3 grid for all cases are shown. Matrices in cases 1 and 2 are symmetric negative definite (all of their eigenvalues are negative). The third case, on the other hand, is indefinite. Nearly half of the eigenvalues are positive, and half of them are negative, corresponding to the left and right sides of the red curve in fig. 15a.

fig. 15b shows the 1-norm approximation of the condition number of the matrices for all cases. Evidently, a larger grid results in a higher condition number. Also, as expected, the condition number in case 2 is higher than case 1, and the condition number in case 3 is higher than case 2.

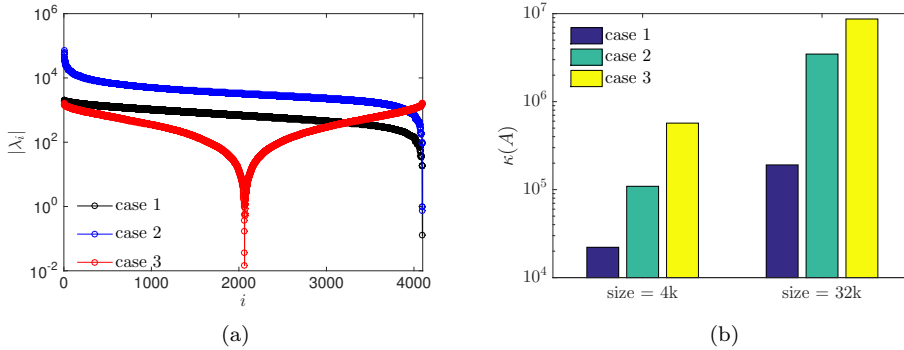


FIG. 15. Properties of the matrices corresponding to the discretization of eq. (6.6): (a) Norm of the eigenvalues for the 16^3 grid. (b) The 1-norm approximation of the condition number for 16^3 and 32^3 grids.

We used LoRaSp as a preconditioner in conjunction with GMRES. A summary of the results for the two first cases is provided in table 1. The convergence criterion of GMRES, with residual defined in eq. (6.5), is set to 10^{-14} . For cases 1 and 2, we used LoRaSp with low-rank precision $\epsilon = 10^{-1}$ as defined in eq. (6.2). Similar to section 6.2.1, we increase the depth of the \mathcal{H} -tree with $\log n$, where n is the size of matrix. Similar to the results presented in section 6.2.1, the factorization time has an almost linear complexity with the size of the matrix. As depicted in fig. 15b, the condition number grows rapidly from case 1 to case 2, and with the size of the matrix. This explains a slight growth of the number of iterations, and relative error.

case	fact. time	GMRES time	tot. time	# iters	rel. error
1 (4k)	0.44	0.06	0.50	10	1.7e-11
1 (32k)	4.62	1.07	5.69	15	2.6e-10
1 (256k)	37.08	16.21	53.29	15	8.6e-10
2 (4k)	0.40	0.07	0.47	12	6.6e-11
2 (32k)	3.72	1.31	5.03	20	9.2e-10
2 (256k)	33.52	19.44	52.95	30	1.2e-9

TABLE 1

GMRES performance using LoRaSp as a preconditioner to solve a variable coefficient Poisson equation. Matrix sizes, corresponding to 16^3 , 32^3 , and 64^3 grids are written in parentheses. Times are reported in seconds.

Case 3 corresponds to an indefinite matrix, with a large condition number. This is typically a more difficult problem, compared to the first two cases. Since case 3 is inherently a harder problem compared to cases 1 and 2, we choose a higher low-rank precision $\epsilon = 10^{-3}$. In fig. 16a, the averaged rank of interactions for each level is plotted. We also plot the compression ratio for each level, which is defined as follows:

$$(6.7) \quad \text{compression ratio } (l) = \frac{\langle \text{interaction rank} \rangle_l}{\langle \text{size of super-nodes} \rangle_l},$$

where $\langle \cdot \rangle_l$ denotes averaging in level l of the \mathcal{H} -tree. Note that it is clear from fig. 16a that even though the rank is increasing, the compression ratio is approximately 0.6–0.7 for all levels. Hence, the algorithm takes advantage of low-rank structures appropriately.

In fig. 16b, the preconditioned GMRES residual defined in eq. (6.5) is plotted as a function of the iteration number. GMRES for case 3 does not converge when no preconditioner or a diagonal preconditioner is used. We also applied the incomplete LU (ILU) factorization [52] as a preconditioner for GMRES. We tried various (including very large) values for the fill parameter in ILU. No convergence was obtained when ILU is used as a preconditioner.

6.2.3. Elasticity equation (unstructured grid). Our next benchmark is obtained from an unstructured grid (see [24]) to solve the three-dimensional elasticity equation:

$$(6.8) \quad (\lambda + \mu)\nabla(\nabla \cdot \mathbf{u}) + \mu\nabla^2 \mathbf{u} + \mathbf{F} = 0.$$

The matrix is symmetric with size $n = 334,956$, and total of 10,977,198 non-zero entries. This problem is significantly more difficult in comparison to the previous benchmarks. We used various preconditioning strategies to evaluate the performance of our proposed solver. In table 2 a summary of the results is provided. We consider 10^{-12} to be the convergence criterion for the GMRES residual, and consider a maximum of 500 iterations.

When no preconditioner is used, GMRES does not converge, and we obtain a solution with 2% relative error after 500 iterations. Employing a diagonal preconditioner (i.e., ignore all non-diagonal entries of the matrix, and approximate A^{-1} with the inverse of its diagonal part) also does not lead to convergence. A 1.4% relative error after 500 GMRES iterations is obtained, which is an improvement in comparison to the case without a preconditioner.

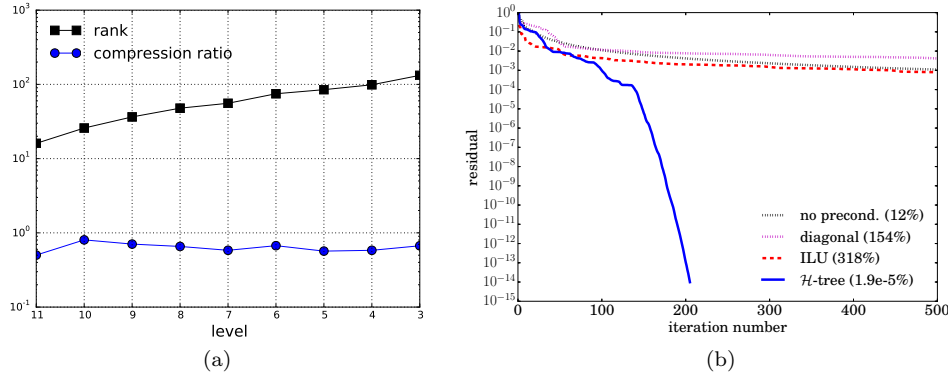


FIG. 16. Results corresponding to the case 3 for a 32^3 grid. (a) Averaged rank and compression ratio in the \mathcal{H} -tree. (b) GMRES residual as a function of the iteration number using various preconditioners. The relative error at the end of the iterations is shown in parentheses for each case. Although the residuals may be small, the relative error for some of the methods is very large.

Next, we tried ILU as a preconditioner for GMRES. We used dual-threshold ILU with drop tolerance fixed equal to the GMRES convergence precision, and varying fill values (1, 2, etc.). ILU with fill value of 1 does not converge after 500 iterations; however, for fill values greater than 1, convergence is obtained before 500 iterations. Increasing the fill value leads to higher factorization time.

Finally, we used the proposed algorithm as a preconditioner. For the low-rank approximation, we used a variation of eq. (6.2). Consider we want to find a low-rank approximation of a block B with singular-value decomposition $B = USV^\top$. We keep the first k singular-values (and therefore, singular vectors) such that, k is the smallest integer that:

$$(6.9) \quad \frac{\|B - U_k S_k V_k^\top\|_F}{\|\text{all levels below}\|_F} < \epsilon.$$

The subscript k in U_k and V_k means keeping only the first k columns, and in S_k means keeping the first k singular-values. $\|\cdot\|_F$ refers to the Frobenius norm. Therefore, $\|\text{all levels below}\|_F$ refers to square root of the sum of Frobenius norm squared of all blocks at the current level as well as the levels below. Both the above criteria and the one in eq. (6.2) work properly, and lead to the same conclusions; however, the above method is slightly more efficient. For this benchmark, we used a sequence of decreasing values for ϵ in eq. (6.9): $\epsilon_1 = 1024 \times 10^{-7}$, $\epsilon_2 = 256 \times 10^{-7}$, $\epsilon_3 = 64 \times 10^{-7}$, $\epsilon_4 = 16 \times 10^{-7}$, $\epsilon_5 = 4 \times 10^{-7}$, $\epsilon_6 = 1 \times 10^{-7}$.

fig. 17 illustrates the variation of the GMRES residual versus the number of iterations using various preconditioners. Clearly, diagonal preconditioner accelerates convergence compared to the case with no preconditioner. ILU preconditioners bring about convergence faster than diagonal. Increasing the fill-value enhances the rate of convergence. The \mathcal{H} -tree based preconditioners lead to a significant acceleration in convergence. Decreasing the low-rank approximations parameter, ϵ , results in faster convergence. In general, decreasing ϵ (and similarly increasing fill value in ILU) results in a shorter iteration time at the cost of a more expensive factorization as listed in table 2. In practice, one should pick an intermediate value for ϵ (and similarly for the

precond.	fact. time	GMRES time	tot. time	# iters	rel. error
none	0	115.9	115.9	500	2.1e-2
diagonal	0.02	116.3	116.3	500	1.4e-2
ILU 1	98.6	156.7	255.3	500	5.3e-06
ILU 2	211.1	132.0	343.1	408	4.9e-10
ILU 3	313.0	102.2	415.2	309	5.7e-10
ILU 4	399.9	82.3	482.2	245	3.2e-10
\mathcal{H} -tree ϵ_1	90.6	298.9	389.5	467	2.3e-10
\mathcal{H} -tree ϵ_2	93.5	299.6	393.1	466	2.2e-10
\mathcal{H} -tree ϵ_3	114.6	295.5	410.1	367	2.4e-10
\mathcal{H} -tree ϵ_4	166.0	130.6	296.6	144	3.5e-10
\mathcal{H} -tree ϵ_5	379.8	74.9	454.7	46	1.3e-10
\mathcal{H} -tree ϵ_6	961.1	43.8	1004.9	16	2.1e-10

TABLE 2

GMRES performance using various preconditioners for a matrix of size 330k with more than 10 million non-zeros obtained from a 3D unstructured discretization of the elasticity equation. Times are reported in seconds.

fill value when ILU is used [52]) to get an optimal total runtime (i.e., factorization + GMRES iterations).

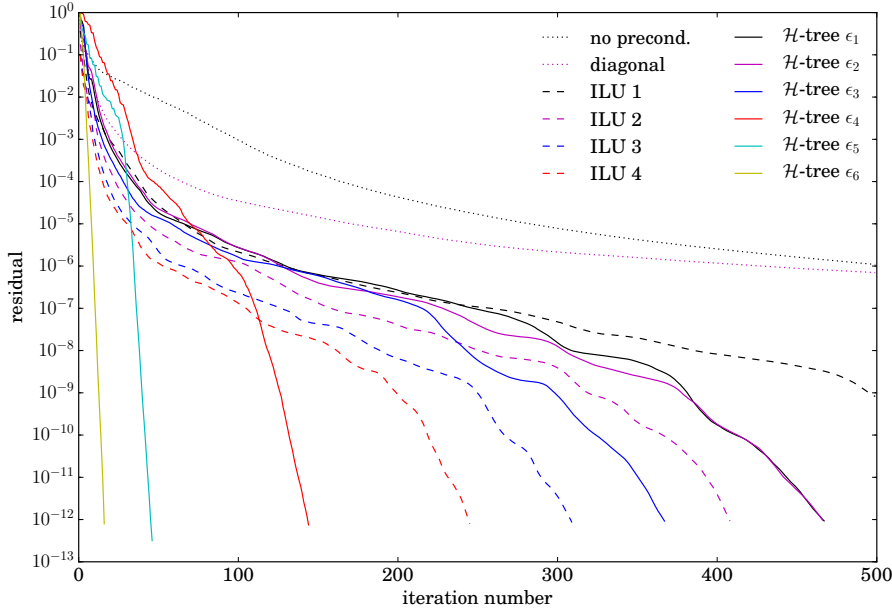


FIG. 17. GMRES residual as a function of iteration number using various preconditioners for a matrix of size 330k with more than 10 million non-zeros obtained from a 3D unstructured discretization of the elasticity equation.

In fig. 18 the breakdown of the total solve time (= factorization time + GMRES time) is plotted for the cases that convergence is achieved. The non-monotonic functionality of the total time as a function of ϵ is clear from this figure. For instance, for this set of ϵ values, $\epsilon_4 = 16 \times 10^{-7}$ has the optimal solve time, which is more efficient

in comparison to the best time of the ILU. Note that the current implementation of the algorithm is completely sequential. There are various optimizations to enhance the performance of the solver. We discuss some of the possible optimizations in section 7. Typically, ϵ meets its optimal value when factorization time and GMRES (or any other iterative method) times are almost equal. This is also evident in fig. 18.

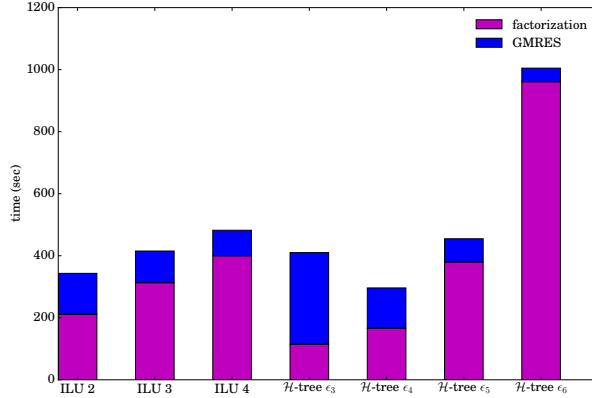


FIG. 18. Total solve time using various preconditioners for a matrix of size 330k with more than 10 million non-zeros obtained from a 3D unstructured discretization of the elasticity equation.

6.2.4. Advection-diffusion problem (non-symmetric linear system).

In our last benchmark, we consider the advection-diffusion problem in a cubic domain of size L^3 with Dirichlet boundary conditions. This problem is governed by the following PDE.

$$(6.10) \quad \frac{\partial T}{\partial t} + \mathbf{U} \cdot \nabla T = \nu \nabla^2 T + s$$

In the above equation $\mathbf{U} = (U_x, U_y, U_z)$ is a constant velocity vector, ν is the diffusion coefficient, and s is a source term. Using an implicit numerical time integration method, and assuming $U_x = U_y = U_z = U$, the above equation transforms to the following non-dimensional form

$$(6.11) \quad \sigma T + \mathcal{R} \nabla T - \nabla^2 T = g,$$

where $\sigma = \frac{L^2}{\nu \Delta t}$ and $\mathcal{R} = \frac{LU}{\nu}$ assuming Δt is the time step.

Discretizing eq. (6.11) using a central finite differencing method results in symmetric and skew-symmetric matrices for the diffusion and advection terms, respectively. Therefore, the full system of equations is represented by a non-symmetric matrix.

We verified the accuracy of the \mathcal{H} -tree solver (similar to fig. 9) for a case with $\sigma = 0$ and $\mathcal{R} = 1$ on a 32^3 grid. fig. 19a shows the error and residual as a function of the low-rank precision parameter ϵ as defined in eq. (6.2). Similar to the symmetric case, residual and error are proportional to ϵ .

Furthermore, we compare the convergence of the GMRES solver for the advection-diffusion problem when \mathcal{H} -tree and ILU are used as preconditioner. We consider a sequence of problems with $\sigma = 1$ and varying \mathcal{R} on a 32^3 grid. In fig. 19b the numbers of GMRES iterations required to converge to a solution with residual less than 10^{-10} are shown for different cases. For the \mathcal{H} -tree preconditioner a low-rank precision of $\epsilon = 10^{-1}$ is used, while for the ILU drop tolerance is equal to the GMRES

convergence precision and fill parameter is set to 3 (the minimum fill parameter such that all cases converge with less than 500 iterations). A 1-norm approximation of the condition number of the system is also illustrated in fig. 19b. For the problem studied here, the condition number (and therefore, the number of GMRES iterations) is a non-monotonic function of \mathcal{R} . \mathcal{H} -tree preconditioner exhibits a stable number of iterations and accuracy for all cases. The ILU preconditioner, however, gives rise to a larger number of iterations and higher variation with \mathcal{R} . Furthermore, for the case with $\mathcal{R} = 1024$ the ILU preconditioner results in a solution with final residual (defined in eq. (6.3)) of order 10^{-2} , while the preconditioned residual (defined in eq. (6.5)) is less than 10^{-10} . Such discrepancy means the preconditioner (in this case ILU) is ill-conditioned.

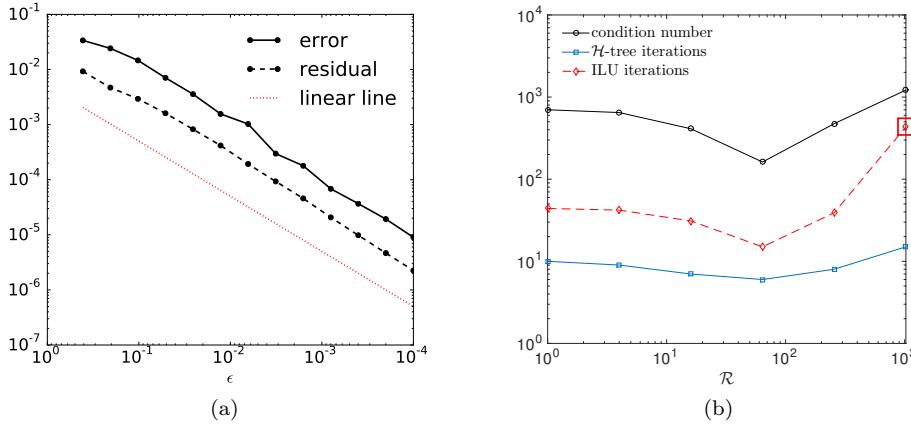


FIG. 19. (a) Error and residual of the solution of eq. (6.11) as a function of low-rank precision ϵ using \mathcal{H} -tree solver. (b) Number of GMRES iterations using ILU and \mathcal{H} -tree as preconditioner. For $\mathcal{R} = 1024$, the ILU preconditioner results in a final residual 10^{-2} , while the pre-conditioned residual is less than 10^{-10} . This case is highlighted by a red square in the figure. The 1-norm approximation of the matrix condition number is also plotted.

7. Conclusion and future works. We proposed a new algorithm to solve sparse linear systems with numerically low-rank structures in linear time. The algorithm is based on the LU factorization of the sparse matrix, where the matrices L and U are computed and stored using a hierarchical low-rank structure. The accuracy of the factorization is determined a priori. For a precision tolerance ϵ , the complexities of the factorization cost and memory are $\mathcal{O}(n \log^2 1/\epsilon)$ and $\mathcal{O}(n \log 1/\epsilon)$, respectively.

The proposed algorithm is fully algebraic and preserves the sparsity pattern of the original matrix during the elimination. Therefore, it can be considered as an extension to the ILU method. In the ILU factorization, new fill-ins are ignored. In the proposed algorithm, however, new fill-ins are compressed using low-rank approximations. Compressed fill-ins form a new set of equations—at a coarser level—which are factorized through elimination similar to the original equations. This recursive process continues until the set of equations in hand is full (but small), and therefore, can be solved directly without introducing new fill-ins. The resulting hierarchy of equations is similar to the \mathcal{H} -tree structure. Furthermore, the multilevel process of the factorization is similar to AMG, where the original system is solved at different levels (grid size). Here, the equations at the coarser level are compressed fill-ins of

the fine level.

We provided various benchmarks to illustrate the performance of the proposed algorithm. We used matrices obtained from the discretization of the Poisson and elasticity equations on structured and unstructured grids, respectively. A non-symmetric benchmark corresponding to the advection-diffusion problem is also presented. The proposed factorization method is used both as a stand-alone solver with tunable accuracy, and as a preconditioner in conjunction with an iterative method (e.g., GMRES).

The algorithm introduced in this paper is based on a general hierarchical framework. There are various aspects of the method which are general, and can be modified to optimize the solver for particular matrices without losing the properties demonstrated in this paper. Here is a list of some aspects that can be modified in the algorithm:

1. Partitioning of the sparse matrix graph is generic. Higher quality partitioning in general results in higher accuracy solution. If the matrix is associated to a physical grid, the physical coordinates of the solution points can be used to improve the quality of partitioning.
2. Here, we used a binary tree corresponding to the recursive bi-partitioning of the graph. Many other options are possible, e.g., using octree if the matrix comes from a three-dimensional problem, or an adaptive tree with arbitrary number of children per node.
3. We used SVD for low-rank representation of the well-separated nodes. Other low-rank approximation methods could be used as well, e.g., randomized SVD [35], randomized block algorithm [55], adaptive cross approximation [8], rank-revealing QR/LU [13], etc.
4. Having a low-rank representation of a well-separated block, there are different measures to define the error. For instance, we used the ratio of the singular values to the largest singular value as a measure of accuracy in the low-rank approximation in eq. (6.2). One can use different criteria, e.g., absolute singular values, ratio of singular values to largest singular value of the whole level or the full matrix, Frobenius norm of the low-rank block, etc.
5. We defined two super-nodes as well-separated based on the distance of their corresponding nodes in the adjacency graph (see definition 3.6). One can change this definition, and make it stronger. For example, define two super-nodes as well-separated if their distance is at least 3 in the adjacency graph. This is similar to the fill value in the incomplete LU factorization.
6. At each level, we can use any ordering to eliminate super-nodes and black-nodes. Here we used a random ordering. There are various other orderings which reduce the calculation cost, including the minimum degree, minimum deficiency, nested dissection, etc. [21]. The complexity of the algorithm remains linear irrespective of the ordering. This is particularly an alluring property for a parallel implementation of the algorithm.

Appendix A. (Factorization process example). In this section, we present an example of the factorization process (see algorithm 1) for one level. Each figure represents one step of the algorithm. The \mathcal{H} -tree is shown on right, and the corresponding extended matrix is shown on the left.

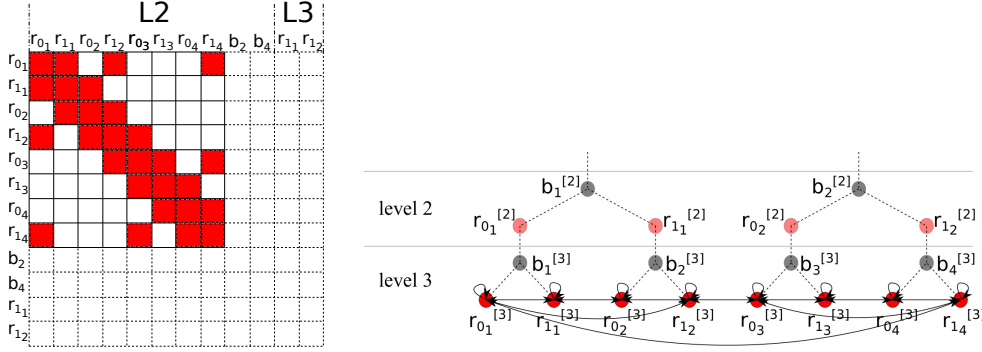


FIG. 20. Original matrix (left) and the corresponding adjacency graph (right). In the left figure, L2 and L3 refers to variables at level 2 and 3, respectively.

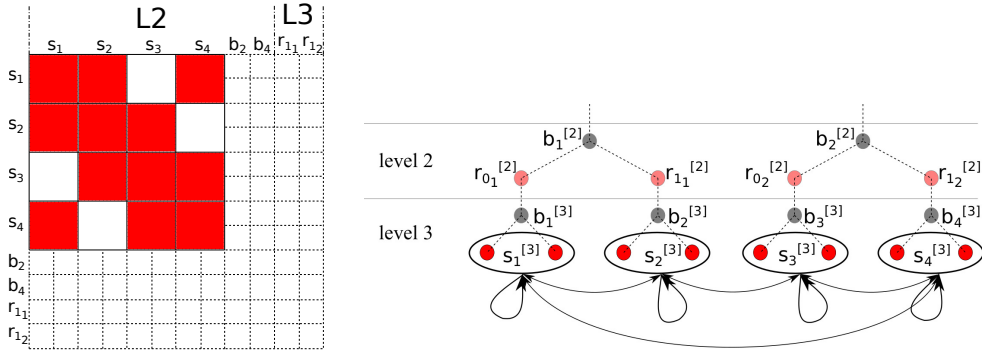


FIG. 21. Creating super-nodes at level 3 (see section 4.2), where the super-node $s_i^{[3]}$ consists of red-nodes $r_{0i}^{[3]}$ and $r_{1i}^{[3]}$.

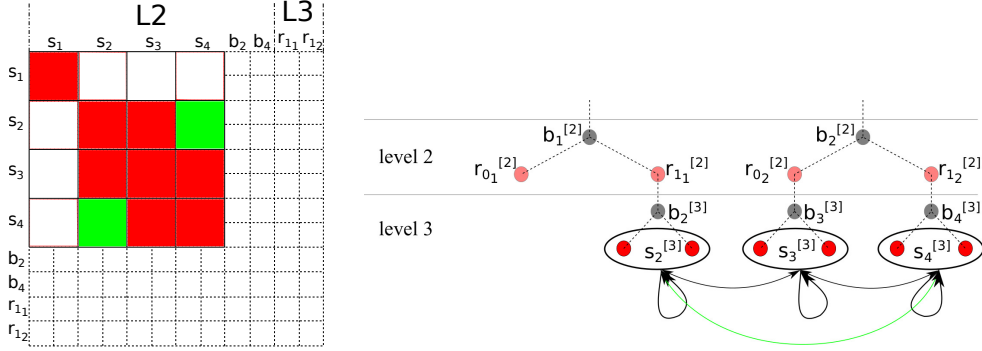


FIG. 22. Eliminating $s_1^{[3]}$ (see section 4.4). Green edges (and their corresponding blocks in the matrix) represent a numerically low-rank interaction between two well-separated nodes to be compressed.

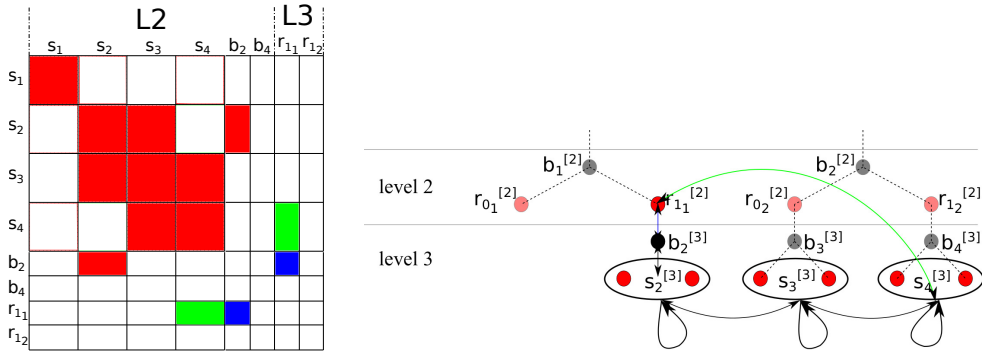


FIG. 23. Compressing the well-separated edge between $s_2^{[3]}$ and $s_4^{[3]}$ (see section 4.3). Blue edges (and their corresponding blocks in the matrix) correspond to minus identity.

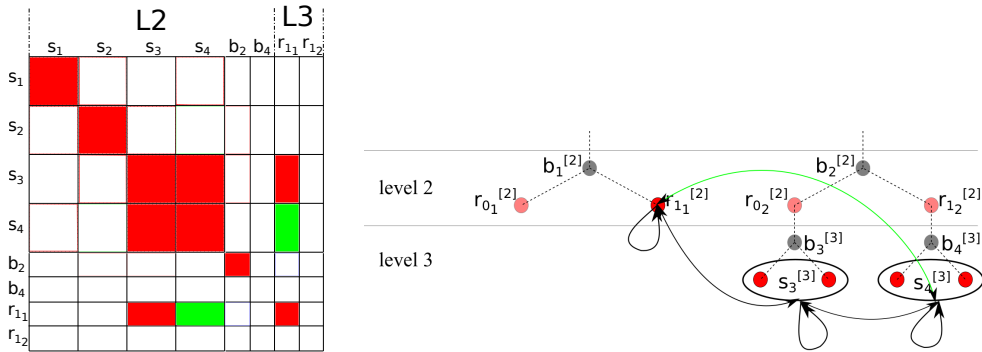
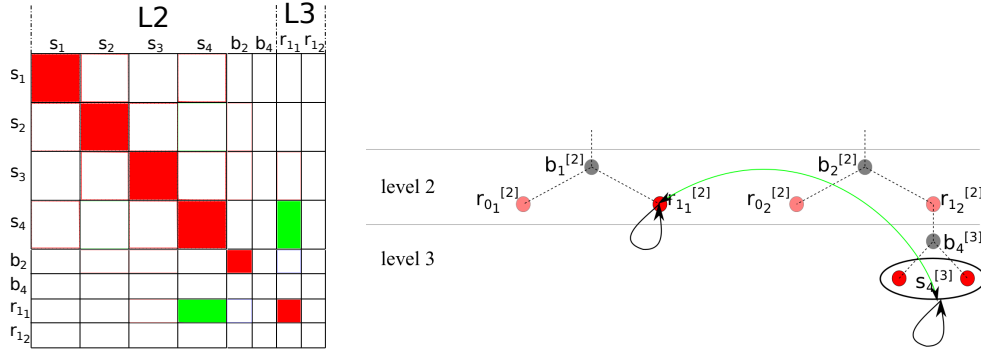
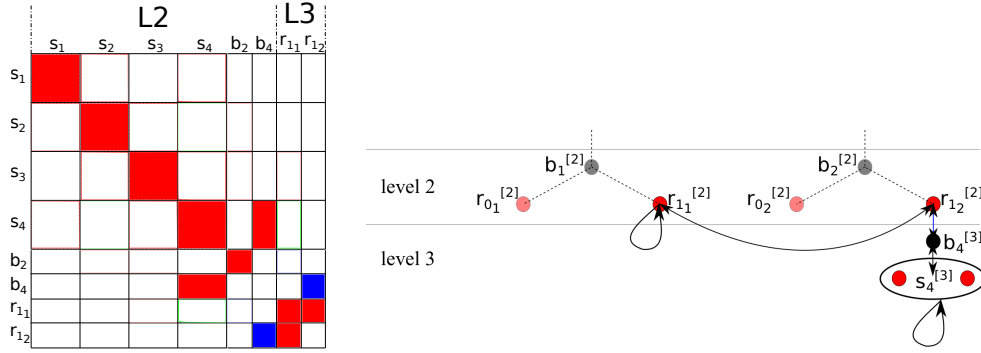
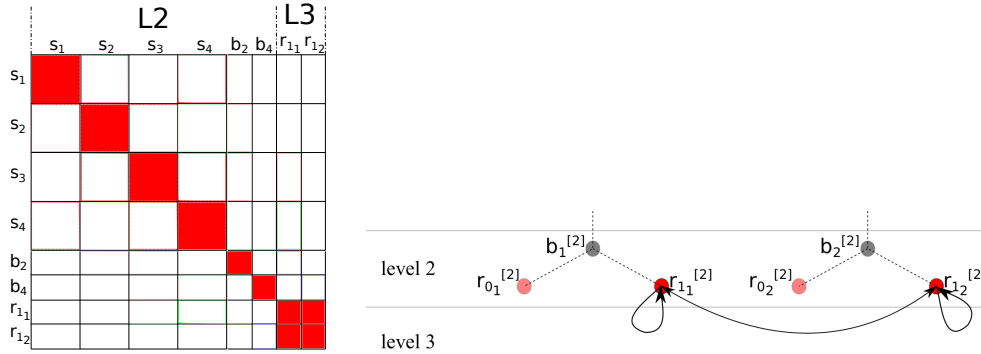


FIG. 24. Eliminating $s_2^{[3]}$ and $b_2^{[3]}$ (see section 4.4).

FIG. 25. Eliminating $s_3^{[3]}$ (see section 4.4).FIG. 26. Compressing the well-separated edge between $r_{11}^{[2]}$ and $s_4^{[3]}$ (see section 4.3).FIG. 27. Eliminating $s_4^{[3]}$ and $b_4^{[3]}$ (see section 4.4).

Appendix B. (Graph partitioning example). In this appendix we provide a graphical example of a nested partitioning using the SCOTCH library for a sparse matrix corresponding to discretization of eq. (6.1) on a 2D Voronoi grid (similar to fig. 8).

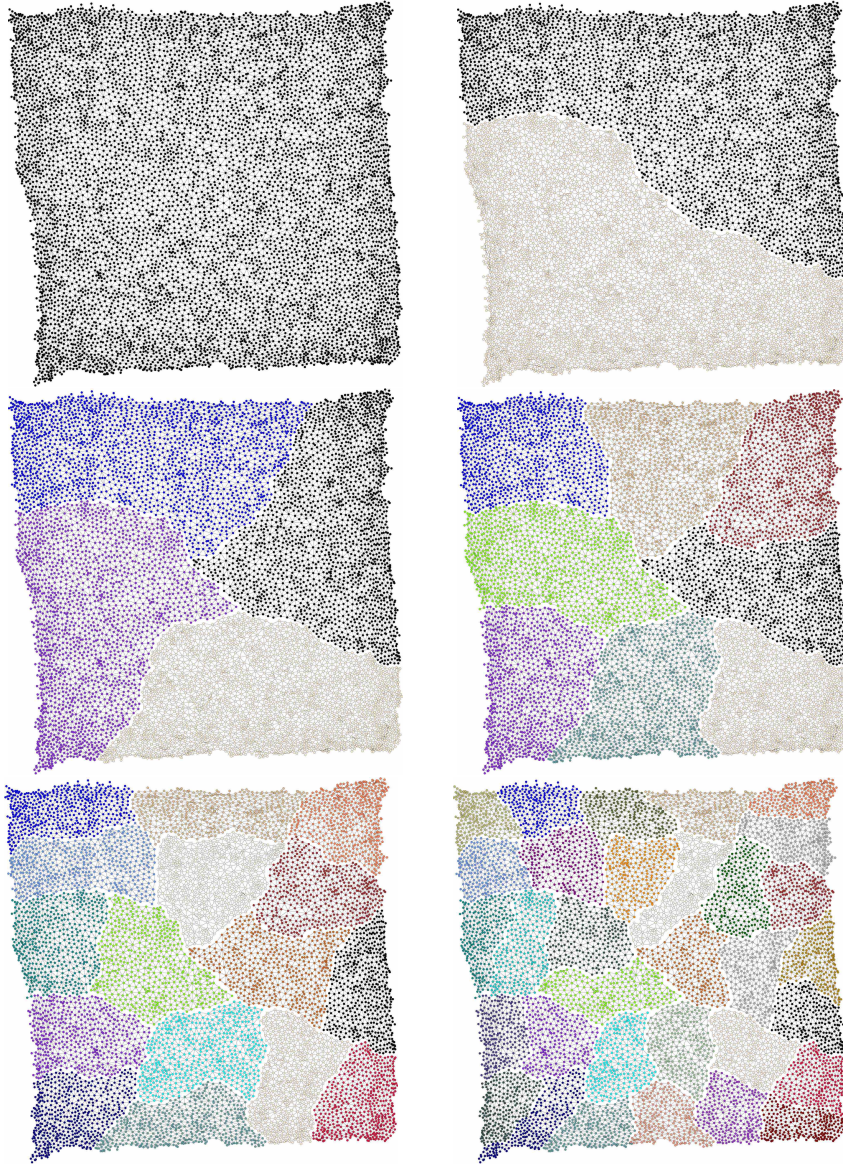


FIG. 28. An example of 6 levels of a nested partitioning. Clusters are distinguished by different colors. The edges between different clusters are intentionally omitted in this figure for visualization purpose.

REFERENCES

- [1] NOGA ALON AND RAPHAEL YUSTER, *Solving linear systems through nested dissection*, in Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on, IEEE, 2010, pp. 225–234.
- [2] SIVARAM AMBIKASARAN, *Fast Algorithms for Dense Numerical Linear Algebra and Applications*, PhD thesis, Stanford University, 2013.
- [3] SIVARAM AMBIKASARAN AND ERIC DARVE, *An $\mathcal{O}(n \log n)$ fast direct solver for partial hierarchically semi-separable matrices*, Journal of Scientific Computing, 57 (2013), pp. 477–501.
- [4] ———, *The inverse fast multipole method*, arXiv preprint arXiv:1407.1572, (2014).
- [5] AMIRHOSSEIN AMINFAR, SIVARAM AMBIKASARAN, AND ERIC DARVE, *A fast block low-rank dense solver with applications to finite-element matrices*, Journal of Computational Physics, 304 (2016), pp. 170–188.
- [6] EDWARD ANDERSON, ZHAOJUN BAI, CHRISTIAN BISCHOF, SUSAN BLACKFORD, JAMES DEMMEL, JACK DONGARRA, JEREMY DU CROZ, ANNE GREENBAUM, S HAMMERLING, ALAN MCKENNEY, ET AL., *LAPACK Users' guide*, vol. 9, Siam, 1999.
- [7] MARIO BEBENDORF, *Hierarchical matrices*, Springer, 2008.
- [8] MARIO BEBENDORF AND SERGEJ RJASANOW, *Adaptive low-rank approximation of collocation matrices*, Computing, 70 (2003), pp. 1–24.
- [9] STEFFEN BÖRM, LARS GRASEDYCK, AND WOLFGANG HACKBUSCH, *Introduction to hierarchical matrices with applications*, Engineering Analysis with Boundary Elements, 27 (2003), pp. 405–422.
- [10] ACHI BRANDT, *Algebraic multigrid theory: The symmetric case*, Applied mathematics and computation, 19 (1986), pp. 23–56.
- [11] A BRANDT, S MCCORUICK, AND J HUGE, *Algebraic multigrid (amg) for sparse matrix equations*, Sparsity and its Applications, (1985), p. 257.
- [12] JAMES R BUNCH AND JOHN E HOPCROFT, *Triangular factorization and inversion by fast matrix multiplication*, Mathematics of Computation, 28 (1974), pp. 231–236.
- [13] TONY F CHAN, *Rank revealing qr factorizations*, Linear algebra and its applications, 88 (1987), pp. 67–82.
- [14] SHIV CHANDRASEKARAN, PATRICK DEWILDE, MING GU, WILLIAM LYONS, AND TIMOTHY PALS, *A fast solver for hss representations via sparse matrices*, SIAM Journal on Matrix Analysis and Applications, 29 (2006), pp. 67–81.
- [15] SHIV CHANDRASEKARAN, MING GU, AND TIMOTHY PALS, *A fast ulv decomposition solver for hierarchically semiseparable representations*, SIAM Journal on Matrix Analysis and Applications, 28 (2006), pp. 603–622.
- [16] Y-H CHOI AND CHARLES L MERKLE, *The application of preconditioning in viscous flows*, Journal of Computational Physics, 105 (1993), pp. 207–223.
- [17] DON COPPERSMITH AND SHMUEL WINOGRAD, *Matrix multiplication via arithmetic progressions*, in Proceedings of the nineteenth annual ACM symposium on Theory of computing, ACM, 1987, pp. 1–6.
- [18] EDUARDO CORONA, PER-GUNNAR MARTINSSON, AND DENIS ZORIN, *An $\mathcal{O}(n)$ direct solver for integral equations on the plane*, Applied and Computational Harmonic Analysis, 38 (2015), pp. 284–317.
- [19] PIETER COULIER, HADI POURANSARI, AND ERIC DARVE, *The inverse fast multipole method: using a fast approximate direct solver as a preconditioner for dense linear systems*, arXiv preprint arXiv:1508.01835, (2015).
- [20] ERIC DARVE, *The fast multipole method: numerical implementation*, Journal of Computational Physics, 160 (2000), pp. 195–240.
- [21] TIMOTHY A DAVIS, *Direct methods for sparse linear systems*, vol. 2, Siam, 2006.
- [22] WILLIAM FONG AND ERIC DARVE, *The black-box fast multipole method*, Journal of Computational Physics, 228 (2009), pp. 8712–8725.
- [23] ALAN GEORGE, *Nested dissection of a regular finite element mesh*, SIAM Journal on Numerical Analysis, 10 (1973), pp. 345–363.
- [24] DEBRAJ GHOSH, PHILIP AVERY, AND CHARBEL FARHAT, *A feti-preconditioned conjugate gradient method for large-scale stochastic finite element problems*, International journal for numerical methods in engineering, 80 (2009), pp. 914–931.
- [25] ADRIANNA GILLMAN AND PER-GUNNAR MARTINSSON, *A direct solver with $\mathcal{O}(n)$ complexity for variable coefficient elliptic pdes discretized via a high-order composite spectral collocation method*, SIAM Journal on Scientific Computing, 36 (2014), pp. A2023–A2046.
- [26] ADRIANNA GILLMAN, PATRICK M YOUNG, AND PER-GUNNAR MARTINSSON, *A direct solver with $\mathcal{O}(n)$ complexity for integral equations on one-dimensional domains*, Frontiers of Mathe-

- mathematics in China, 7 (2012), pp. 217–247.
- [27] LESLIE GREENGARD, DENIS GUEYFFIER, PER-GUNNAR MARTINSSON, AND VLADIMIR ROKHLIN, *Fast direct solvers for integral equations in complex three-dimensional domains*, Acta Numerica, 18 (2009), pp. 243–275.
 - [28] LESLIE GREENGARD AND VLADIMIR ROKHLIN, *A fast algorithm for particle simulations*, Journal of computational physics, 73 (1987), pp. 325–348.
 - [29] GAËL GUENNEBAUD, BENOÎT JACOB, ET AL., *Eigen v3*. <http://eigen.tuxfamily.org>, 2010.
 - [30] HERVÉ GUILLARD AND CÉCILE VIOZAT, *On the behaviour of upwind schemes in the low mach number limit*, Computers & fluids, 28 (1999), pp. 63–86.
 - [31] WOLFGANG HACKBUSCH, *A sparse matrix arithmetic based on h-matrices. part i: Introduction to h-matrices*, Computing, 62 (1999), pp. 89–108.
 - [32] WOLFGANG HACKBUSCH AND STEFFEN BÖRM, *H2-matrix approximation of integral operators by interpolation*, Applied Numerical Mathematics, 43 (2002), pp. 129–143.
 - [33] WOLFGANG HACKBUSCH AND BORIS N KHOROMSKIY, *A sparse h-matrix arithmetic: general complexity estimates*, Journal of Computational and Applied Mathematics, 125 (2000), pp. 479–501.
 - [34] WILLIAM W HAGER, *Condition estimates*, SIAM Journal on Scientific and Statistical Computing, 5 (1984), pp. 311–316.
 - [35] NATHAN HALKO, PER-GUNNAR MARTINSSON, AND JOEL A TROPP, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM review, 53 (2011), pp. 217–288.
 - [36] MAGNUS RUDOLPH HESTENES AND EDUARD STIEFEL, *Methods of conjugate gradients for solving linear systems*, (1952).
 - [37] NICHOLAS J HIGHAM AND FRANÇOISE TISSEUR, *A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra*, SIAM Journal on Matrix Analysis and Applications, 21 (2000), pp. 1185–1201.
 - [38] KENNETH L HO AND LEXING YING, *Hierarchical interpolative factorization for elliptic operators: differential equations*, Communications on Pure and Applied Mathematics, (2015).
 - [39] ———, *Hierarchical interpolative factorization for elliptic operators: integral equations*, Communications on Pure and Applied Mathematics, (2015).
 - [40] OSCAR H IBARRA, SHLOMO MORAN, AND ROGER HUI, *A generalization of the fast lup matrix decomposition algorithm and applications*, Journal of Algorithms, 3 (1982), pp. 45–56.
 - [41] WAI YIP KONG, JAMES BREMER, AND VLADIMIR ROKHLIN, *An adaptive fast direct solver for boundary integral equations in two dimensions*, Applied and Computational Harmonic Analysis, 31 (2011), pp. 346–369.
 - [42] RUIPENG LI AND YOUSEF SAAD, *Divide and conquer low-rank preconditioners for symmetric matrices*, SIAM Journal on Scientific Computing, 35 (2013), pp. A2069–A2095.
 - [43] RICHARD J LIPTON, DONALD J ROSE, AND ROBERT ENDRE TARJAN, *Generalized nested dissection*, SIAM journal on numerical analysis, 16 (1979), pp. 346–358.
 - [44] ARTEM NAPOV AND XIAOYE S LI, *An algebraic multifrontal preconditioner that exploits the low-rank property*, Numerical Linear Algebra with Applications, 23 (2016), pp. 61–82.
 - [45] NAOSHI NISHIMURA, *Fast multipole accelerated boundary integral equation methods*, Applied Mechanics Reviews, 55 (2002), pp. 299–324.
 - [46] IVAN V OSELEDETS AND SV DOLGOV, *Solution of linear systems and matrix inversion in the tt-format*, SIAM Journal on Scientific Computing, 34 (2012), pp. A2718–A2739.
 - [47] CHRISTOPHER C PAIGE AND MICHAEL A SAUNDERS, *Solution of sparse indefinite systems of linear equations*, SIAM journal on numerical analysis, 12 (1975), pp. 617–629.
 - [48] FRANÇOIS PELLEGRINI AND JEAN ROMAN, *Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs*, in High-Performance Computing and Networking, Springer, 1996, pp. 493–498.
 - [49] HADI POURANSARI AND ERIC DARVE, *Optimizing the adaptive fast multipole method for fractal sets*, SIAM Journal on Scientific Computing, 37 (2015), pp. A1040–A1066.
 - [50] HADI POURANSARI, MILAD MORTAZAVI, AND ALI MANI, *Parallel variable-density particle-laden turbulence simulation*, Annual Research Briefs, Center for Turbulence Research, (2015), pp. 43–54.
 - [51] JW RUGE AND KLAUS STÜBEN, *Algebraic multigrid*, Multigrid methods, 3 (1987), pp. 73–130.
 - [52] YOUSEF SAAD, *Ilut: A dual threshold incomplete lu factorization*, Numerical linear algebra with applications, 1 (1994), pp. 387–402.
 - [53] YOUSEF SAAD AND MARTIN H SCHULTZ, *Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on scientific and statistical computing, 7 (1986), pp. 856–869.
 - [54] KLAUS STÜBEN, *A review of algebraic multigrid*, Journal of Computational and Applied Math-

- ematics, 128 (2001), pp. 281–309.
- [55] SERGEY VORONIN AND PER-GUNNAR MARTINSSON, *A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices*, arXiv preprint arXiv:1503.07157, (2015).
 - [56] JIANLIN XIA, SHIVKUMAR CHANDRASEKARAN, MING GU, AND XIAOYE S LI, *Fast algorithms for hierarchically semiseparable matrices*, Numerical Linear Algebra with Applications, 17 (2010), pp. 953–976.
 - [57] MIHALIS YANNAKAKIS, *Computing the minimum fill-in is np-complete*, SIAM Journal on Algebraic Discrete Methods, 2 (1981), pp. 77–79.
 - [58] LEXING YING, GEORGE BIROS, AND DENIS ZORIN, *A kernel-independent adaptive fast multipole algorithm in two and three dimensions*, Journal of Computational Physics, 196 (2004), pp. 591–626.